SGS-THOMSON MICROELECTRONICS

APPLICATION NOTE

REAL-TIME KERNELS ON THE ST20

by Julian Wilson

This note is a preliminary look at the facilities provided by the ST20 architecture for implementing a real-time kernel.

1 WHAT IS A KERNEL?

A real-time kernel is a mechanism for controlling the execution of the tasks existing in a system. A real-time system has timing constraints governing its behaviour which mean that some actions must be carried out by particular deadlines. An example of real-time is the controlling of a fuel valve to a motor, an example of non real-time is the updating of a record in a wages data base.

The important aspect of controlling the execution of tasks is that of scheduling. A real-time system must be able to perform a particular action in a particular time and thus the kernel must be able to guarantee that the task will be scheduled in a specifiable time. In other words the most important feature of a real-time kernel is that it MUST be DETERMINISTIC.

The most significant aspect of the micro-kernel built into the ST20 architecture is that it is NON-DETERMINISTIC. This is so because a process is scheduled onto the back of a queue of unknown length. In many embedded systems this non-determinism does not matter, what matters is that the overall work rate keeps up with the actual data rate. In some environments, however, it matters a great deal and in these cases the ST20 micro kernel must be enhanced.

A deterministic system is one in which the time taken from the occurrence of an event and the software associated with that event running on the processor has a known maximum duration. If the maximum duration is less than the required time for the application the system should be able to support the application. A standard mechanism for providing deterministic scheduling is to implement a kernel which supports multiple levels of priority in which a higher priority task will always preempt a lower priority task and the time for preemption is known. Preemption is the act of suspending the current task and starting the new task.

This note is concerned with the requirements for implementing a multi-priority preemptive scheduler on the ST20.

Table of Contents

1 WHAT IS A KERNEL? 1
2 WHAT MUST IT DO? 3
3 HOW CAN IT DO IT? 3
4 THE ST20 4
4.1 TRAPS 6
4.2 INTERRUPTS 8
5 IMPLEMENTATION ON THE ST20 9
5.1 EXTENSION 10
5.1.1 Implementation 12
5.1.1.1 Queue empty trap 16
5.1.1.2 All other traps 18
5.2 REPLACEMENT 20
5.2.1 Implementation 21
5.2.1.1 Queue empty trap 22
5.2.1.2 Timeslice traps 24
5.2.1.3 All other traps 26
5.3 ST20 HIGH PRIORITY PROCESSES 27
6 PERFORMANCE 28
7 CONCLUSION 28
APPENDIX 29

2 WHAT MUST IT DO?

A multi-priority preemptive scheduler has a number of jobs to perform. These are:

– Keep track of all user tasks in the system.
– Keep track of all system resources.
– Make certain that the highest priority runnable task is actually the task which is running.
– Provide a set of user accessible task-management functions.
– Share the available CPU time fairly between tasks with the same priority.
– Fail to a known safe state.
– Recover from failure in a safe and controlled manner.
– Protect itself and other user's tasks from errant behaviour in user's tasks.
– Manage mutual exclusion.
– Manage access to shared resources.
– Additionally functions the kernel should endeavour to perform include.
– Use the CPU efficiently.
– Make certain that the response to external stimuli occurs as quickly as possible.
– Provide visibility of state so that debug and monitoring tools can be implemented easily.
– Manage priority enhancemet for access to shared resources.

It can be seen that any kernel which attempts to do the above jobs must be involved in every aspect of the functioning of the CPU. In particular, the kernel must be aware of every scheduling act which takes place.

3 HOW CAN IT DO IT?

A real-time system is a collection of interacting objects which have particular characteristics. Some of these objects are tasks. A task will have characteristics such as:

– Priority
– Resources owned
– Current instruction pointer
– Current stack pointer
– Current state
– Child tasks
– Parent tasks
– Memory range

and so on.

It is convenient to keep this collection of attributes in a data structure in an accessible place. The structure is usually refered to as a PD (Process Descriptor) and the kernel

uses the set of PDs to manage the tasks and to hold all current information about them. The set of PDs is normally kept as a linked list which has a head and tail pointer. Tasks can be added to and removed from the list using standard creation and destruction functions. The PD is the repository of all information about a task and so is effectively the task's ID.

Other objects exist as part of a kernel. Examples are queues, resources, trap handlers, interrupt handlers etc. All of these objects will also have some kind of of management structure. The structures which describe the objects in the system are used by the kernel to control the actions requested on any object. For example, a request to kill a task would require knowledge of the task's current state, which resources it currently owns, whether the request is legal, which other tasks are dependent on the task to be killed and so on. All of this information must be available through the kernel's object data structures.

It is normal to talk of a kernel as if it is a separate task running in the system which performs actions independently. This is not, normally, the case. The kernel is, usually, a set of functions which are called directly or indirectly by the tasks themselves. As shall be shown in the next section, the ST20 is somewhat unusual in this area. The functions which constitute the kernel perform jobs such as task creation/destruction, queue insertion and descriptor maintenance. Many of these functions require priviliges and must be completed as a single action to guarantee no invalid transient information in the system data structures.

To re-cap, a real-time kernel is a set of data structures and functions which provide a mechanism for controlling access to the resources of a system in a predictable and deterministic manner.

4 THE ST20

The ST20 microprocessor is a member of the ST20 family of application targetted microprocessors which have an advanced modular architecture. The individual members are created by combining a CPU core with an application orientated set of macro cells to produce an overall processor which is a custom fit to the chosen application. The CPU cores range from very small simple and low power where the primary emphasis is on very small silicon area to high performance where the CPU may have significant work to do. The ST20 is a specific variant of the architecture with a mid-range performance level targetted at the requirements for the set-top-box market.

The CPU core in the ST20 is the C4 core which includes a number of autonomous scheduling capabilities. The C4 core has a micro-kernel which maintains two process queues; a low-priority queue which shares CPU time between the processes on the

queue by time-slicing and a high-priority queue which supports processes which must run to completion. The processor has a single serial link, a set of prioritised vectored interrupts, two timers and several external DMA engines which all of which can autonomously schedule processes onto the process queues.

The native scheduling model implemented by the ST20 is a cooperative non-preemptive model. The processes on the low-priority queue use the CPU until they become blocked or until their timeslice slot has expired. If their timeslice slot has expired they are put on the back of the low-priority queue and the process on the front of the queue is resumed from the point at which it was last descheduled. Processes which become ready to run (as a result of an external stimulus or some action by the currently running process) are added to the back of the queue ready to take their turn on the CPU. High-priority processes run until they become blocked, they do not time-slice and they cannot be preempted. High-priority processes always preempt low priority process and run until they cannot continue.

An interrupt on a traditional microprocessor will, usually, cause some scheduling actions to occur. On the ST20 this mechanism is significantly extended with autonomous scheduling occurring as the result of the following events.

– timeslice
– input message
– output message
– timer expiration
– process stopping
– semaphore wait
– semaphore signal
– traps and interrupts

The first seven actions place a process on the end of one of the scheduling queues and may also cause a context switch. The cooperative scheduling model supported allows a relatively large number of light weight processes to share available CPU resource very efficiently. When a process on the low-priority queue gives up the processor if it has become blocked or if its timeslice period has expired the next process on the queue is automatically scheduled by the hardware with no software intervention. This hardware scheduling results in a very low context-switch time of, approximately, 500 nanoseconds. The time until a process actually runs from when it is placed on the queue is unknown, however, because the queue is of unknown length. If the target system demands that this time must be less than a fixed maximum or that there must be a hierarchy between the processes active on the system then it may be necessary to implement an alternative scheduling scheme. The simplest solution to this non-determinism problem is by a clear understanding of the process model of the

system, using high-priority processes for time critical actions and knowing the maximum number of processes whic may exist on the low-priority queue at any moment. For more complex systems such a detailed knowledge may be infeasible.

Alternative scheduling models must be implemented by a software kernel which determines which process to run and manages the cpu queues. A software kernel on the ST20 may be implemented as an extension to the ST20 micro kernel or as a complete alternative to the ST20 micro kernel. Which of these two approaches is chosen largely depends on the features and performance required from the kernel to be implemented.

The architecure of the ST20 includes some features to assist in constructing a kernel with a richer scheduling model. These features are:

– traps for all scheduling actions
– trap on queue empty
– timeslice enable/disable
– interrupt enable/disable
– force timeslice
– queue manipulation instructions for
  queue insertion
  shadow register manipulation
  process restart
  save and restore queue pointers
  swap run queue
  swap timer queue

The basic mechanism to assist the kernel writer is the scheduling trap. Every time a process is about to be added to the high or low priority run queue one of the scheduling traps will be invoked if it is enabled.

4.1 TRAPS

The ST20 has four trap groups for both high and low priority, the groups are Breakpoint, Error, System and Scheduler. The scheduler group is different from the others in that it does not trap exceptions but intercepts actions of the ST20 micro kernel. The traps in the scheduler group are QueueEmpty, External Channel, Internal Channel, Timer, TimeSlice, Run, Signal and ProcessInterrupt. QueueEmpty indicates that there is no more work to do and all of the others indicate that the hardware scheuler is about to schedule a process on a queue.

Trap handlers and trapped processes are set up and examined via the four instructions:

1. ldtraph load trap handler structure

2. sttraph store trap handler structure
3. ldtrapped load trapped process structure
4. sttrapped store trapped process structure

A scheduler trap is invoked when a process is about to be run. The trapped process is the process which has been interrupted and the trap handler is the process which is invoked to handle the trap. The trapped process and the trap handler process data structure are used to pass information to the handler, to save state and to reset the state of the processor as it enters the trap handler. The handler structure Enables word is used to reset the trap and global interrupt enable state of the processor and is typically set up to prevent another trap from the same group occurring while the trap handler is running to avoid reentrancy problems. The trapped process structure contains the trap bit which indicates the source of the trap. The structures are as follows:

Trap handler structure		
Enables	Mask word which will be ANDed with the trap enable and global interrupt bits. Bits 15 - 0 are the trap bits mask Bits 23 - 0 are the global interrupt mask	Base + 0
Status	Trap handler initial status register value	Base + 1
Wptr	Trap handler initial stack pointer	Base + 2
Iptr	Start of trap handler code	Base + 3

Trapped process structure		
Enables	Enables register bits at time of trap	Base + 0
Status	Status register at time of trap	Base + 1
Wptr	Trapped task current stack pointer	Base + 2
Iptr	Next instruction to execute on return from trap	Base + 3

The process descriptor (stack pointer + priority) of the process about to be scheduled is passed to the trap handler in traphandler Stack[0]. For the scheduler traps, other than the queue empty trap, the trap handler has details of the process which has been suspended, and the process about to be scheduled and can do one of the following:

1. Priority (New) < Priority (Trapped)
   Save new task on appropriate suspended queue.
   Return to trapped task.
2. Priority (New) = Priority (Trapped)
   Add new process to back of current processor run queue.
   Return to trapped task.
3. Priority (New) > Priority (Trapped)

Suspend current processor run queue, saving state of trapped task.
Put new task on processor run queue.
Invalidate trapped process state.
Return from trap (will select first task on the processor run queue).

For the queue empty trap the trap handler must suspend the current queue and run the highest priority non-empty queue. If there is no queue available with processes to run, the queue empty trap must be disabled to prevent continuous queue empty traps occurring.

The intention of the scheduler traps is to allow a real-time kernel writer to extend the capabilities of the built-in micro-kernel to implement more powerful scheduling facilities yet, at the same time, use the micro-code wherever possible to deliver very fast context switching.

The trap handler invoked is the trap handler for the priority of the process about to be scheduled rather than the process currently executing. Enabling scheduler traps only really makes sense at low priority to implement software managed scheduling. Within this environment high priority processes are much more akin to interrupt handlers or pieces of hardware and should be used for time critical functions, DMA handlers (e.g.links) and for watchdog processes.

For many applications a combination of ST20 native high-priority processes and real-time kernel scheduled low-priority tasks produces a very-efficient and flexible solution to real-world scheduling requirements. The example schedulers given in Appendices B and C demonstrate this model.

*******to be extended to describe all state when a scheduler trap occurs********

4.2 INTERRUPTS

The ST20 supports eight prioritized external interrupts which may be configured to be at a higher or lower priority than the high priority process queue. Each interrupt level has a single software handler which is initiated by the on-chip interrupt controller if the associated interrupt pin has been activated and the interrupt is enabled. An interrupt handler always pre-empts a scheduled process (i.e. not a trap handler) and any interrupt handler at a lower priority (0 = lowest to 7 = highest).

An interrupt handler is set up by writing its WPtr (stack pointer) to the appropriate HandlerWptr register, setting the trigger mode in the TriggerMode register and then setting the matching enable bit in the Mask register. The trigger mode is flexible and can be set to be edge or level and high or low. In common with standard ST20 processes the interrupt handler's entry point is stored in and retrieved from its stack (which is unlike a trap handler).

5 IMPLEMENTATION ON THE ST20

The primary decision when implementing a software scheduler on the ST20 is whether to use the hardware scheduler but extend its capabilities or replace the hardware scheduler completely. The answer to this depends largely on what other features the new scheduling scheme requires. If the new scheduling scheme is intended just to provide prioritized scheduling then a relatively lightweight modification of the built-in scheduler is probably the right solution. If, however, the scheduling scheme is part of a more complete real-time kernel then it may be desirable to replace the microkernel completely.

The decision as to which type of scheduler to use is also governed by the characteristics of the target application. A typical embedded application creates a set of tasks with a fixed relationship to one another, starts them and leaves them to run until power-off. Embedded applications normally have no dynamic characteristics nor any sharing of resources between tasks. In such a static environment, the cost incurred when doing some out of the ordinary action, such as killing another task, is an acceptable penalty for the benefit of very fast lightweight context switching during normal running.

Typical facilities offered by more ambitious real-time kernels include facilities to manage system resources, kill other tasks, promote or decay the priority of tasks, manage memory, manage time slicing and use different scheduling algorithms from round robin with time-slicing.

An example of the complexity of the internals of a real-time kernel is an environment in which task T0 of priority P0 wishes to run but cannot because it needs a resource R0. R0 however is currently owned by task T1 of priority P1 where P1 is lower than P0. A third task, T2 of priority P2 where P2 is higher than P1 but lower than P0, is running. As a result T0 is being blocked by the lower priority task T2. Similarly intricate interconnections occur when tasks attempt to kill other tasks.

To manage a kernel of any complexity it is necessary to to maintain data structures for all of the objects controlled by the kernel. The data structure for managing tasks is usually called a Process Descriptor (PD). The PD of any task is the repository of all information on that task and is the handle by which any scheduling type activity is managed.

The lists of object descriptors, especially PDs, are used by the kernel to manage the system. It is imperative to the kernel that it is in control at all times and is aware of the complete state of the processor. Any autonomous scheduling activity by the hardware must inform the kernel so that it can keep all state information complete and consistent. This is partly achieved on the ST20 by enabling the scheduler traps. Any

autonomous scheduling action to place a process on the run queue is intercepted and the ST20 traps to a handler when a process is about to be added to one of the queues and presents it with details of the process about to be scheduled.

The stack pointer of the about to be scheduled process is passed to the trap handler in the trap handler's stack. The trap handler must determine what to do with the newly runnable task, whether to run it, queue it on the processor's run queue or place it on a currently inactive queue. How to determine the scheduling priority of the new task is the major difficulty for the kernel implementer.

Descheduling actions, such as waiting on a semaphore or timer do not generate traps, however, and, as a consequence, the current PD is invalidated without the kernel being informed. This is a serious problem for a more general kernel which is discussed in more detail in section 5.2.

The following two sections describe a scheduling algorithm which is a lightweight extension to the hardware scheduler and a more general scheduling mechanism which is suitable as the basis for building a real time operating system.

5.1 EXTENSION

This section looks at the mechanisms which might be employed to implement a preemptive scheduler but still allow the built-in micro-kernel to do as much of the scheduling work as possible.

The first decisions to make are what are the goals of the scheduler. For this section they are as follows:

– Multiple priorities.
– Pre-emptive scheduling.
– Minimum context switch time.
– Hardware scheduling between tasks of the same priority.

The fourth goal, to use the built-in micro kernel when scheduling between tasks of the same priority, has a profound effect on the way the scheduler must work. The minimum scheduling time occurs when the micro kernel time slices between the tasks at a particular priority with the timeslice trap disabled. Whenever a new task is about to be scheduled and causes a trap the trap handler must decide if the task is higher, lower or equal in priority to the tasks on the current queue.

If the new task is lower in priority it is added to the appropriate inactive queue and the currently executing task is resumed. If the new task is equal in priority it is added to the back of the CPU active queue and the currently executing task is resumed. If the new task is higher in priority the current queue must be suspended and a new queue started with the new task as the only runnable task.

The critical action is determining the relative priority of the about to be scheduled task. The trap provides the workspace descriptor (stack pointer + ST20 priority) of the new process but no other information. To determine the scheduling priority the stack pointer must be mapped to some data which gives details about the task. One possibility would be to compare address ranges against a task list and identify the task by its stack bounds. Any form of searching is not acceptable in a real-time environment because it will be very time-consuming for all but the simplest systems and very cumbersome if stacks can be created and destroyed dynamically.

When a task is descheduled by the ST20, information is stored at small negative offsets from the workspace pointer as follows:

Offset	Name	Function
-1	Iptr	address of next instruction to execute
-2	Link	link to next process on this queue
-3)	Pointer	pointer to communication data area
-3)	State	saved state
-4	Tlink	link to next process on timer queue (if appropriate)
-5	Time	time process is waiting for (if appropriate)

The stack allocated to each task must accomodate these extra 5 words. A simple scheduler could be implemented easily by adding another element to this data structure containing the task's priority, i.e.

-6	Priority	scheduling priority of process

Implementing a scheduler becomes a matter of making sure that this slot is valid whenever it needs to be. It does not need to be valid when the task is active or runnable because the priority can be inferred from the priority of the current queue, it must be valid when the task is not runnable. The intructions which may cause a task to be unable to continue are:

1. altwt wait for one of many messages
2. in input message
3. out output message
4. outbyte output byte
5. outword output word
6. stopp stop process
7. taltwt timed wait for one of many messages
8. wait wait on semaphore

Making certain that the workspace -6 slot is valid when any of these 8 instructions is executed enables the trap handler to determine the priority of the process when it is about to be rescheduled. The current running priority can be maintained in a global location. The normal method of forcing the location to be valid is is to allow access to

these instructions only via library calls which copy the global running priority value into the designated stack slot. When the hardware micro-kernel timeslices a process it is moved to the back of the current run queue and no intervention is necessary.

5.1.1 Implementation

Appendix B is an example implementation of a lightweight scheduler referred to as OS/20 Lite. Also available are routines to replace all standard library functions which may cause a deschedule. The library replacement routines write the current priority into Stack[-6] immediately before executing the instruction which may deschedule. The number of priorities supported is user definable - the example is set at 8.

A sample user application is included in Appendix D. The user task initializes the scheduler and then starts and runs the application tasks. After starting all of the application tasks the main user task enters a wait loop waiting for all of the application tasks to terminate. As each application finishes it returns to a common task exit routine which signals a semaphore before terminating.

Preemption performance is measured by a lower-priority tasks scheduling a higher priority task using a semaphore. The lower priority task samples the time immediately before the signal and the higher priority task samples the time immediately on exiting the wait. The difference is thus the time from executing code in a lower priority task to executing code in a higher priority task including the signal and wait. This time is approximately 6 microseconds.

The scheduling is managed using a set of task queues, each of which corresponds to the ST20 process queue. The state for each queue is maintained in a structure consisting of a front pointer, a back pointer, a boolean and a block of state information. The front and back pointers are workspace pointer values to be placed in the ST20 CPU front and back pointers, the boolean indicates whether the queue had been interrupted and that the state data should be restored when the queue is restarted. The kernel maintains a single item of dynamic state - the priority of the current queue which is used to access all other state information.

When an ST20 is running a process, that process is not present on the processor queue, its state is the state of a number of registers in the CPU. The processor queue is a linked list of all of the processes which can run but are waiting for CPU time and has a front pointer to the next process to be run and a back pointer to the place to add newly active processes. The complete active process state thus consists of the current state of the CPU plus the queue head and tail pointers. The CPU current state consists of 12 words:

– Enables flags

– Status register

– Workspace pointer
– Instruction pointer
– Stack registers (Areg, Breg and Creg)
– 2D Block move registers (5 words)

The state data is a copy of the trapped process information which is in the same format as the shadow registers. The queue structure is thus an exact copy of the information the ST20 needs to restore a previous environment. The task queue structure is defined in os20lite.h as

```
typedef struct task_queue
{
 WORKSPACE q_fptr;
 WORKSPACE q_bptr;
 BOOL    q_suspended;
 STATE   q_pstate;
}QUEUE;
```

using the definitions:

```
typedef unsigned long   UINT32;
typedef unsigned long*   WORKSPACE;
typedef struct cpu_stack_s
{
 UINT32   c_areg;
 UINT32   c_breg;
 UINT32   c_creg;
}CPU_STACK;
typedef struct cpu_state_s
{
 UINT32   c_enables;
 UINT32   c_status;
 WORKSPACE  c_wptr;
 void*    c_iptr;
}CPU_STATE;
typedef struct processor_state
{
```

```
 CPU_STATE  s_cpu_state;
 CPU_STACK  s_cpu_stack;
 UINT32   s_move2d[NO_MOVE2D_WORDS];
}STATE;
```

The first 4 words of the processor state (the enables bits, status register, workspace pointer and instruction pointer) is the state of the CPU at the time of interrupt and is identical to the contents of the trapped process structure.

When a scheduler trap occurs the trap handler needs information about the running process which has been interrupted and the new process which has caused the trap. Information about the interrupted process is written by the CPU into a reserved area of memory. This information can be accessed by pointing directly at the reserved memory area or it can be copied onto the trap handler's stack using the sttrapped instruction or as follows:

```
__asm
 {
  ldabc   SCHEDULER_TRAP, &TrappedTask, priority_low;
  sttrapped;
 }
```

where Trapped Task is declared on the local stack as

```
trap_structure_t TrappedTask;
```

The trapped task structure returned by the sttrapped instruction is defined in the standard header file rmclib.h. It is the same structure as that used when installing a trap handler using the instruction ldtraph and is defined as

```
typedef struct trap_structure_s
    {
     enables_word_t   enables_word;
     status_register_t  status_word;
     void        *wptr;
     union
     {
      void *address;
      void (*handler_fn)(int /*Areg*/, int /*Breg*/, int /
       *Creg*/);
```

```
 }          iptr;
} trap_structure_t;
```

The cause of the trap is determined by examining the status word in the above structure which is a copy of the status register and is defined in rmclib.h as

typedef union status_register_u

```
{
 struct
 {
  unsigned int trap_reason          : 14;
  unsigned int unknown1            : 1;
  unsigned int status_trap_causeerror     : 1;
  unsigned int scheduler_trap_priority    : 2;
  unsigned int trap_group           : 2;
  unsigned int timeslice_enable        : 1;
  unsigned int unknown2            : 5;
  unsigned int interrupted_operation_code   : 5;
  unsigned int process_valid         : 1;
 }  bits;
 unsigned int word;
} status_register_t;
```

and is tested by comparing the 14 bit trap_reason field against the enables bits as follows

if (TrappedTask.status_word.bits.trap_reason == enable_trap_queue_empty)

When a scheduler trap is taken the contents of the CPU stack and the workspace descriptor of the task being scheduled are written onto the trap handler's stack. The trap handler accesses these details via the trap_state_t structure which is pointed to by the compiler predefine _params (which is a mechanism for allowing any function to access its own context regardless of the number of local variables declared). _params is declared at the start of the trap handler as

```
extern const volatile trap_state_t* _params;
```

The trap_state structure is defined in rmclib.h as

```
typedef struct trap_state_s
{
```

```
    void *wdesc;
    void *gsb;
    int  Areg;
    int  Breg;
    int  Creg;
  } trap_state_t;
```

The first field in this structure wdesc is the process descriptor of the process about to be scheduled while the three fields Areg, Breg and Creg are a copy of the processor stack at the time the trap was taken and thus belong with the trapped task rather than the task causing the trap.

The scheduler traps of interest to OS/20-Lite are the queue empty (processor idle) trap and all traps caused by an attempt to queue a process, such as may happen on signal or after a timed wait expires. OS/20-Lite does not enable timeslice traps.

5.1.1.1 Queue empty trap

The queue empty trap indicates that all of the processes on the current queue are no longer active and a new, lower priority, queue should be selected.

The example OS/20-Lite implementation selects the next queue by a simple search down the list from the current priority. This algorithm is suitable for relatively few priority levels (say 8) but may not be suitable for a high number of priority levels. Another mechanism may be to keep the active queues in a linked list with a pointer to the next queue to run. The disadvantage of the second algorithm is the time taken to insert the queue into the list, which is time taken from a higher priority task whereas the simple algorithm costs time only after the higher priority task has released the processor. Which mechanism is chosen is largely dependent on the system required. A system with a large number of priorities (say 256) but with only a few active at a time would benefit from the linked list whereas a system with a few priorities (say 8) would be most efficiently implemented as a simple search.

If an active queue is found the CPU run queue must be loaded with the front and back pointer from the saved queue. If the new active queue had previously been interrupted by a higher priority queue, the CPU state must be restored from the saved state vector. The swapqueue instruction replaces the current front and back run queue pointers and the restart instruction restores all of the state but does not clear the trap reason from the saved status register. (The {{{ and }}} markers in the code are artifacts of the folding editor used to create the code and are comments when the file is seen by the compiler.)

```
if (Queue->q_suspended)
```

```
 {{{ restore the preempted queue
 {
  STATE* State = &Queue->q_pstate;
  {{{ set CPU front and back pointers to values in task queue
       table
  __asm
   {
    ldabc priority_low, Queue->q_fptr, Queue->q_bptr;
    swapqueue;
   }
  }}}
  Queue->q_suspended = FALSE;
  State->s_cpu_state.c_status &= CLEAR_TRAP_REASON_MASK;
  __asm
   {
    ld  &Queue->q_pstate;
    restart;
   }
 }
 }}}
else if (Queue->q_fptr != (WORKSPACE)QUEUE_EMPTY)
 {{{ select a queue and exit
 {
  __asm
   {
    ldabc priority_low, Queue->q_fptr, Queue->q_bptr;
    swapqueue;
    tret;
   }
 }
 }}}
```

If no active queue is found the idle trap is disabled to prevent continuous traps being taken:

TrappedTask->enables_word &= (~enable_trap_queue_empty);

5.1.1.2 All other traps

When any trap occurs other than the queue-empty trap, a task has become ready to run and must be added to a queue. The new task is at a lower, equal or higher priority than the current priority and so must: be queued on a waiting queue; be added to the CPU run queue or preempt the current queue. The priority of the new task is found by examining the slot in the negative workspace:

```
WORKSPACE NewWptr   = (WORKSPACE)((int)(_params->wdesc) &
       (~PRIORITY_MASK));
int    NewPriority = (int)NewWptr[PW_PRIORITY];
```

Lower priority

Adding to a lower priority queue is simply a case of updating the appropriate queue pointers:

```
{{{ add to lower priority queue
{
 QUEUE* Queue = TaskQueue[NewPriority];
 if (Queue->q_fptr != (WORKSPACE)QUEUE_EMPTY)
  Queue->q_bptr[PW_LINK] = (unsigned int)NewWptr;
 else
  Queue->q_fptr = NewWptr;
 Queue->q_bptr  = NewWptr;
}
}}}
```

Equal priority

Adding to the same priority queue is done by putting the task at the back of the current processor queue:

```
{{{ add to cpu run queue
{
 __asm
  {
   ld   _params->wdesc;
```

```
   runp;
  }
}
}}}
```

Higher priority

If the new task is at a higher priority than the current priority, the current task and CPU state must be saved to the appropriate queue and the new task run. Saving the current task is performed by collecting and storing the 12 state words as follows:

```
State = &(CurrentQueue->q_pstate);
{{{ save state of current task
__asm
 {
  ldabc   MOVE2D_SAVE_MASK, priority_low, State;
  stshadow;     /* save block move state */
  ldabc   ABC_SIZE_IN_BYTES, &State->s_cpu_stack, &Areg;
  move;       /* copy A, B and C from Params */
  ldabc   SCHEDULER_TRAP, &State->s_cpu_state, priority_low;
  sttrapped;    /* copy trapped task state */
 }
}}}
```

CurrentQueue->q_suspended = TRUE;

Loading the new queue is a combination of initialising the CPU run queue and marking the trapped process as invalid. The swapqueue instruction leaves the previous values on the CPU stack so that they can be saved into the appropriate place in the data structure:

```
{{{ load new front and back pointers and save old
__asm
 {
  ldabc  priority_low, NewWptr, NewWptr;
  swapqueue;
  stab  CurrentQueue->q_fptr, CurrentQueue->q_bptr;
 }
```

```
}}}
TrappedTask.status_word.bits.process_valid = FALSE;
CurrentPriority = NewPriority;
```

The mechanisms described in this section deliver a fast lightweight multi-priority preemptive scheduler which uses the micro coded queue manipulation facilities to implement timeslicing between tasks of the same priority. If the kernel needs to be involved whenever a task switch occurs a more intrusive scheduling regime must be implemented. An example of such a mechanism follows in the next section.

5.2 REPLACEMENT

OS/20-Lite is suitable when very lightweight and fast scheduling is desired. A more complicated real-time kernel has a great deal more to manage than just the scheduling and must determine which task to run at all times. Typically such a kernel cannot allow any autonomous scheduling actions by the hardware scheduler. A task now needs to be identified by a Task Descriptor and it is this Descriptor which must be stored at workspace slot -6 to identify the task during a scheduling action. It is also necessary to know the state of each task all of the time so that it can be manipulated as needed.

It is desirable to be able to implement such a kernel by enabling the timeslice trap and to allow the micro-kernel to manage the run queue but trapping into the kernel on scheduling activity, timeslice and queue empty. The problem is that some scheduling actions, such as timeslice, trigger a trap and some, such as wait, do not. To accomodate this either two separate mechanisms must be used to manage the tasks or the library routines for wait, in, out etc would need to simulate a scheduling trap. Traps can be forced using the causeerror instruction but complicate the trap handler and quite often will be unnecessary because the instruction may not cause a deschedule.

To avoid the dilemma, an alternative approach, and one favoured by developers porting an existing real-time kernel is to run with an empty run queue. All scheduling traps, the timeslice trap and the queue empty (which is really processor idle) trap are enabled. Any action by the CPU to remove the process from the processor causes a queue-empty trap, any action to schedule a process causes a scheduler trap and a timeslice causes a timeslice trap. All scheduling activity, thus, causes a trap which can be monitored by the kernel.

In this environment the running process continues until:

it is blocked and a queue empty trap occurs

a scheduling trap is taken for another task

the timeslice period expires

The kernel now has complete control of all scheduling and is aware of every state change and the state of every managed object in the system.

5.2.1 Implementation

Appendix C is an example implementation of a scheduler referred to as OS/20. Also available are routines to replace all standard library functions which may cause a deschedule. The replacement routines write the current task into Stack[-6] immediately before calling the instruction which may deschedule.

The sample user application described for OS/20-Lite and included in Appendix D is the same for both OS/20 and OS/20-Lite. One difference from OS/20-Lite is that the wait task has a task structure created for it so that it is a normal O/20 task which can be queued and dequeued like any other. The preemption performance is measured in an identical manner and is approximately 6 microseconds. Timeslice performance of OS/20 is, obviously, much worse than that of OS/20-Lite.

The scheduling is managed using a set of task queues. The tasks are linked in a simple list with a head and tail pointer, a boolean, a block of state information and a task pointer. The front and back pointers are pointers to task structures, the boolean indicates whether the queue had been interrupted and that the state data should be restored when the queue is restarted and the additional task pointer is a pointer to the data structure for the suspended task. The scheduler maintains two items of dynamic state, a current task as well as a current priority but it is the current task which is written to Stack[-6] when a task is descheduled.

The ST20 CPU run queue is not used at all, all task switching is achieved by reading from and writing to the trap structure so that the CPU returns from the trap directly to the task selected to run after the trap. In OS/20-Lite if a new task was to be executed as a result of a trap, the trap structure was modified to indicate that the current task was no longer valid and the new task was added to the front of the CPU run queue.

The decision not to use the CPU queues frees the implementor from the need to maintain the task queue as a linked list of workspace pointers using the PW_LINK slot (-2) of the workspace. Two sample implementations are provided. The first implementation removes each process from the queue when it becomes the active process putting it onto the back of the queue when it is timesliced. The second implementation maintains a circular list, updating the front and back pointers when a timeslice occurs. Not surprisingly, the first implementation is faster at preemption but slower at timeslicing. The remainder of this section refers to the first of these two implementations. The queue and task structures are defined in os20.h as

```
typedef struct queue_s
```

```
{
 TASK*    q_front;
 TASK*    q_back;
 BOOL     q_suspended;
 STATE    q_pstate;
 TASK*    q_task;
}QUEUE;
typedef struct task_s
{
 WORKSPACE    t_wptr;       /* current workspace pointer
       */
 void*      t_iptr;       /* current instruction pointer
       */
 struct task_s* t_link;       /* link to next task in this
       queue */
 int       t_id;        /* id of this task */
 int       t_priority;     /* task priority */
 TASK_STATE   t_state;       /* current state */
 unsigned int*  t_stack_base;    /* pointer to base of stack
       */
 int       t_stack_size;    /* size of stack in bytes */
}TASK;
```

The q_pstate state vector is identical to that described for for OS/20-Lite:

Access to information about the running process which has been interrupted, the new process which has caused the trap and the reason for the trap is obtained in exactly the same manner as before.

The scheduler traps of interest to OS/20 are the queue empty (processor idle) trap, the timeslice trap and all traps caused by an attempt to queue a process, such as may happen on signal or after a timed wait expires.

5.2.1.1 Queue empty trap

The queue empty trap indicates that all of the processes on the current queue are no longer active and a new, lower priority, queue should be selected.

The example OS/20 implementation selects the next queue by maintaining an active queue register with a bit set for each active queue. The highest order set bit is found by use of the norm instruction. If an active queue is found the CPU must be loaded with either the task at the front of the queue or the task which was running when the queue was preeempted.

Resumption after preemption is implemented as follows:

```
{{{ restore the preempted process
{
 STATE* State = &Queue->q_pstate;
 CurrentTask     = Queue->q_task;
 Queue->q_suspended = FALSE;
 if (Queue->q_front == (TASK*)QUEUE_EMPTY)
  QueueActive ^= 1 << CurrentPriority;
 State->s_cpu_state.c_status &= CLEAR_TRAP_REASON_MASK;
 __asm
  {
   ld  State;
   restart;
  }
}
}}}
```

Selecting a new task from the front of the queue:

```
{{{ remove and run only the front process
{
 TASK* Front = Queue->q_front;
 CurrentTask  = Front;
 Queue->q_front = Front->t_link;
 if (Queue->q_front == (TASK*)QUEUE_EMPTY)
  QueueActive ^= 1 << CurrentPriority;
 {{{ copy Front details into TrappedTask structure
 TrappedTask->status_word.bits.process_valid = TRUE;
 TrappedTask->wptr              = (int*)Front->t_wptr;
```

```
 TrappedTask.->iptr.address          = (void*)Front-
       >t_iptr;
 }}}
 __asm {tret;}
}
}}}
```

If no active queue is found the idle trap is disabled to prevent continuous traps being taken:

5.2.1.2 Timeslice traps

When a timeslice trap occurs it may be necessary to replace the current task. If so, the only state which must be saved is the current workspace pointer and the current instruction pointer as all other possible state is not valid at a timeslice point. The timeslice trap code thus links the current task onto the back of the current queue and takes a new task from the front of the queue as the current task.

A refinement on this algorithm would be to allocate tasks multiple timeslice slots according to a priority or need scheme and only replace the task when the allocated number of timeslice periods has expired. A scheme which allocated a percentage of CPU time according to priority could be implemented in this manner.

If no other task is available at the current priority the trap handler just exits back to the interrupted task.

```
{{{ replace the current task
{
 QUEUE* Queue = TaskQueue[CurrentPriority];
 if (Queue->q_front != (TASK*)QUEUE_EMPTY)
  {
   TASK*  OldTask = CurrentTask;
   TASK*  NewTask;
   {{{ put current task at back of queue
   OldTask->t_wptr    = (WORKSPACE)TrappedTask.wptr;
   OldTask->t_iptr    = (void*)TrappedTask.iptr.address;
   OldTask->t_link    = (TASK*)QUEUE_EMPTY;
   Queue->q_back->t_link = OldTask;
   Queue->q_back     = OldTask;
```

```
   }}}
   {{{ access front of queue and write to trapped task struc-
       ture
   NewTask          = Queue->q_front;
   TrappedTask->wptr     = (int*)NewTask->t_wptr;
   TrappedTask->iptr.address = NewTask->t_iptr;
   Queue->q_front       = NewTask->t_link;
   CurrentTask        = NewTask;
   }}}
  }
}
}}}
```

Priority modification

It is sometimes desirable to increase or decrease the priority of the executing task. This is achieved in OS/20 using the timeslice trap. The priority in the task structure is modified to the new value and a timeslice is forced. On receipt of the timeslice trap the trap handler either updates the current priority or dequeues the current task onto a lower priority. Because no tasks are on the CPU queue increasing the priority consists of nothing more than updating the current priority value.

```
int TaskPriority = CurrentTask->t_priority;
if (TaskPriority == CurrentPriority)
 ... replace the current task
else if (TaskPriority > CurrentPriority)
 {{{ task is raising its priority - just reset current priority
 CurrentPriority = TaskPriority;
 }}}
else
 {{{ task is lowering its priority - put on back of appropriate
       queue
 {
  QUEUE* Queue = TaskQueue[TaskPriority];
  TASK*  OldTask = CurrentTask;
  OldTask->t_wptr = (WORKSPACE)TrappedTask.wptr;
```

```
  OldTask->t_iptr = (void*)TrappedTask.iptr.address;
  OldTask->t_link = (TASK*)QUEUE_EMPTY;
  if (Queue->q_front != (TASK*)QUEUE_EMPTY)
   Queue->q_back->t_link = OldTask;
  else
   Queue->q_front = OldTask;
  Queue->q_back = OldTask;
  QueueActive |= 1 << CurrentPriority;
  TrappedTask->status_word.bits.process_valid = FALSE;
 }
 }}}
__asm {tret;}
```

5.2.1.3 All other traps

When any trap occurs other than the above two traps, a task has become ready to run and must be added to a queue. The new task is at a lower, equal or higher priority than the current priority and so must: be queued on a waiting queue or preempt the current queue. The priority of the new task is found by examining the data in the task structure which has been obtained from the slot in the negative workspace:

```
WORKSPACE  NewWptr;
TASK*    NewTask;
NewWptr   = (WORKSPACE)((int)(_params->wdesc) &
       (~PRIORITY_MASK));
NewTask   = (TASK*)NewWptr[PW_TASK];
NewPriority = NewTask->t_priority;
```

Lower or equal priority

Adding to a lower priority queue is simply a case of updating the appropriate queue pointers:

```
{{{ add to lower priority queue
{
 QUEUE* Queue = TaskQueue[NewPriority];
 NewTask->t_wptr = NewWptr;
 NewTask->t_iptr = (void*)NewWptr[PW_IPTR];
 NewTask->t_link = (TASK*)QUEUE_EMPTY;
```

```
 if (Queue->q_front != (TASK*)QUEUE_EMPTY)
  Queue->q_back->t_link = NewTask;
 else
  Queue->q_front  = NewTask;
 Queue->q_back  = NewTask;
 QueueActive |= 1 << NewPriority;
}
}}}
```

Higher priority

If the new task is at a higher priority than the current priority, the current task and CPU state must be saved to the appropriate queue and the new task run. Saving the current task is performed by collecting and storing the 12 state words as shown for OS/20-Lite with the addition of the requirement to save the current task and update the queue active bit.

```
CurrentQueue->q_task = CurrentTask;
QueueActive |= 1 << CurrentPriority;
Loading the new task is done by resetting the global values and
       writing to the trapped process structure.
CurrentPriority = NewPriority;
CurrentTask   = NewTask;
{{{ reset status bits in TrappedTask structure
TrappedTask->wptr     = (int*)NewWptr;
TrappedTask->iptr.address = (void*)NewWptr[PW_IPTR];
}}}
__asm {tret;}
```

The mechanisms described in this section deliver much of the functionality needed for building a full featured real-time operating system taking advantage of the features provided by the ST20. Because both the ST20 state and the OS/20 state is small the context switching time is kept well inside 10 microseconds which compares favourably with proprietary real-time kernels on other microprocessors.

5.3 ST20 HIGH PRIORITY PROCESSES

The ST20 maintains a separate queue for high and low priority processes. Processes on the high priority queue run until they are blocked, they do not timeslice and cannot

be pre-empted (except by higher priority interrupt handlers as described earlier). Processes on the low priority queue share the CPU using a round-robin timeslicing scheme and will always be pre-empted by high priority processes and interrupt handlers. The mechanisms described in this application note are designed to implement a multi priority scheduling regime on the low priority queue only.

The high priority queue may be used as a completely separate scheduling mechanism to provide an exceptionally low latency pre-emption mechanism for extremely time critical actions. The low to high pre-emption time is of the order of 600 nanoseconds (compared with the 5 - 7 microseconds of OS/20). It must, however, be remembered that high priority processes cannot be usurped from the CPU so should be used in much the same manner as interrupt handlers on other microprocessors. A high priority process would typically wake up, grab some data, signal a worker process that data is available then go back to sleep. Using this model and one of the multi-priority scheduling schemes described an extremely powerful and flexible, application sympathetic kernel can be constructed.

The trap mechansim as implemented disables all interrupts while the trap handler is running.

6 PERFORMANCE

7 CONCLUSION

This note has shown that the combination of the ST20 micro scheduler and the scheduler traps enable a highly efficient application sympathetic kernel to be constructed with very low cost. Such a kernel has a number of advantages over proprietary real-time kernels.

no royalty payment

very fast context switch (<6 microseconds)

completely customisable to application requirements

small size

hardware / software ratio can be adjusted as desired

ST20 toolset compatible

available in source form

Where the scheduling needs of the application cannot be met by the native cooperative scheduling algorithm of the ST20 the facilities provided by OS/20 offer a realistic alternative

APPENDIX

A. Details of available state information when within a trap handler.

B. Appendix B

```
{{{ Title
/*************************************************************
 *                                                           *
 *   Module   : os20lite.c                                   *
 *   Purpose  : ST20-TP1 kernel (OS/20-Lite) scheduler trap  *
 *          handler                                          *
 *   Author   : Julian Wilson December 1995                  *
 *                                                           *
 *************************************************************/
}}}
{{{ includes
#include <stdlib.h>
#include <semaphor.h>
#include <rmclib.h>
#include ”os20lite.h”
#include ”library.h”
}}}
{{{ globals
int* CurrentPriorityPtr;
}}}
{{{ statics
PRIVATE Semaphore  TaskComplete;
PRIVATE int     ActiveTasks = 0;
PRIVATE TASK*    TaskList[MAX_TASKS];
PRIVATE int     TaskCount;
}}}
{{{ test globals
```

```
int HigherSchedules = 0;
int EqualSchedules  = 0;
int LowerSchedules  = 0;
int IdleSchedules   = 0;
int NewQueues     = 0;
int Idles       = 0;
int Resumes      = 0;
}}}
{{{ functions
{{{ void* internal_memory_allocate (size_t Requested)
PRIVATE void* internal_memory_allocate (size_t Requested)
{
  PRIVATE unsigned int  Address  = MEMSTART;
  PRIVATE int       Available = RESERVED_INTERNAL;
  void*          Memory   = NULL;
  int   Claimed;
  if (Available > Requested)
   {
     Memory  = (void*)Address;
     Claimed  = (Requested + (BYTES_PER_WORD - 1)) & WORD_MASK;
     Address += Claimed;
     /*printf ("Claimed %d Bytes up to address %8X\n", Claimed,
        Address - 1);*/
   }
  return Memory;
}
}}}
{{{ void kernel_scheduler (int Areg, int Breg, int Creg)
{{{ tell compiler this is the trap handler
void kernel_scheduler (int Areg, int Breg, int Creg);
#pragma IMS_trap_handler (kernel_scheduler)
}}}
```

```
void kernel_scheduler (int Areg, int Breg, int Creg)
{
 extern const /*volatile*/ trap_state_t* _params;
 {{{ declare and place stacktics (on stack statics)
 QUEUE**      TaskQueue;
 trap_structure_t*  TrappedTask;
 int         CurrentPriority;
 #pragma IMS_place_at_workspace_offset (TaskQueue,    9)
 #pragma IMS_place_at_workspace_offset (TrappedTask,   8)
 #pragma IMS_place_at_workspace_offset (CurrentPriority, 7)
 }}}
 if (TrappedTask->status_word.word & enable_trap_queue_empty)
  {{{ find next highest priority active queue
  {
   QUEUE*  Queue;
   for (--CurrentPriority; CurrentPriority >= 0; CurrentPri-
       ority--)
    {
     Queue = TaskQueue[CurrentPriority];
     if (Queue->q_suspended)
      {{{ restore the preempted queue
      {
       STATE* State = &Queue->q_pstate;
       /*Resumes++;*/
       {{{ set CPU front and back pointers to values in task
       queue table
       __asm
        {
         ldabc priority_low, Queue->q_fptr, Queue-
       >q_bptr;
         swapqueue;
        }
```

```
        }}}
        Queue->q_suspended = FALSE;
        State->s_cpu_state.c_status &=
        CLEAR_TRAP_REASON_MASK;
        __asm
         {
          ld  &Queue->q_pstate;
          restart;
         }
       }
       }}}
      else if (Queue->q_fptr != (WORKSPACE)QUEUE_EMPTY)
       {{{ select a queue and exit
       {
        /*NewQueues++;*/
        __asm
         {
          ldabc priority_low, Queue->q_fptr, Queue-
        >q_bptr;
          swapqueue;
          tret;
         }
       }
       }}}
     }
    Idles++;
    {{{ disable idle trap
    TrappedTask->enables_word &= (~enable_trap_queue_empty);
    }}}
   }
   }}}
  else
```

```
{{{ schedule the process onto a queue
{
 int      NewPriority;
 WORKSPACE   NewWptr;
 NewWptr   = (WORKSPACE)((int)(_params->wdesc) &
     (~PRIORITY_MASK));
 NewPriority = (int)NewWptr[PW_PRIORITY];
 if (NewPriority > CurrentPriority)
  {{{ replace current queue if there is one
  {
   /*HigherSchedules++;*/
   if (CurrentPriority < 0)
    {{{ cpu was idle, enable idle trap and load new queue
    {
     TrappedTask->enables_word |=
     enable_trap_queue_empty;
     {{{ load new front and back pointers
     __asm
      {
       ldabc  priority_low, NewWptr, NewWptr;
       swapqueue;
      }
     }}}
    }
    }}}
   else
    {{{ replace current queue
    {
     QUEUE* CurrentQueue = TaskQueue[CurrentPriority];
     STATE* State     = &(CurrentQueue->q_pstate);
     {{{ save current state
     __asm
```

```
    {
     #if 0
     ldabc   MOVE2D_SAVE_MASK, priority_low, State;
     stshadow;                /* save shadow
   information */
     #endif
     ldabc   ABC_SIZE_IN_BYTES, &State->s_cpu_stack,
   &Areg;
     move;                   /* copy A, B and C
   from Params */
     ldabc   SCHEDULER_TRAP, &State->s_cpu_state,
   priority_low;
     sttrapped;    /* copy trapped task state */
    }
   }}}
   {{{ load new front and back pointers and save old
   __asm
    {
     ld   NewWptr;
     dup;
     ld   priority_low;
     swapqueue;
     stab  CurrentQueue->q_fptr, CurrentQueue-
   >q_bptr;
    }
   }}}
   TrappedTask->status_word.bits.process_valid = FALSE;
   CurrentQueue->q_suspended  = TRUE;
  }
  }}}
 {{{ CurrentPriority = NewPriority;
 __asm
  {
```

```
      ld   NewPriority;
      st   CurrentPriority;
     }
    }}}
    __asm {tret;}
   }
   }}}
  else if (NewPriority == CurrentPriority)
   {{{ add to cpu run queue
   {
    /*EqualSchedules++;*/
    __asm
     {
      ld   _params->wdesc;
      runp;
     }
   }
   }}}
  else
   {{{ add to lower priority queue
   {
    QUEUE* Queue = TaskQueue[NewPriority];
    /*LowerSchedules++;*/
    if (Queue->q_fptr != (WORKSPACE)QUEUE_EMPTY)
     Queue->q_bptr[PW_LINK] = (unsigned int)NewWptr;
    else
     Queue->q_fptr = NewWptr;
    Queue->q_bptr  = NewWptr;
   }
   }}}
 }
 }}}
```

```
 {{{ registers for trapped task
 __asm
  {
   ldabc  Areg, Breg, Creg;
  }
 }}}
}
}}}
{{{ BOOL kernel_start (int PriorityLevels)
BOOL kernel_start (int PriorityLevels)
{
 TASK*  Task;
 QUEUE* TaskQueue;
 QUEUE** TaskQueuePtr;
 int   StartupPriority = PriorityLevels - 1;
 {{{ malloc space for and intialise queues and lists
 {
  int i;
  TaskQueuePtr = internal_memory_allocate (sizeof(QUEUE*) *
       PriorityLevels);
  TaskQueue   = internal_memory_allocate (sizeof(QUEUE) *
       PriorityLevels);
  for (i = 0; i < PriorityLevels; i++)
   {
    TaskQueuePtr[i]      = &TaskQueue[i];
    TaskQueue[i].q_fptr    = (WORKSPACE)QUEUE_EMPTY;
    /*TaskQueue[i].q_active   = FALSE;*/
    TaskQueue[i].q_suspended = FALSE;
   }
  Task = internal_memory_allocate (sizeof(TASK) * MAX_TASKS);
  for (i = 0; i < MAX_TASKS; i++)
   TaskList[i] = &Task[i];
```

```
}
}}}
{{{ initialise global state
TaskCount = 0;
SemInit (&TaskComplete, 0);
}}}
{{{ create trap handler environment and install it
{
 int* HandlerWorkspace;
 HandlerWorkspace = internal_memory_allocate (TRAP_WS_SIZE);
 if (HandlerWorkspace == NULL)
  return FALSE;
 install_trap_handler (trap_group_scheduler,
             priority_low,
             (enables_word_t)EVERYTHING_DISABLED,
             default_status_register,
             HandlerWorkspace,
             TRAP_WS_SIZE,
             kernel_scheduler);
 HandlerWorkspace[TRAP_WS_WORDS - (HANDLER_PARAMS + 1)] =
      (int)TaskQueuePtr;
 HandlerWorkspace[TRAP_WS_WORDS - (HANDLER_PARAMS + 2)] =
      (int)LOW_TRAPPED_TASK;
 HandlerWorkspace[TRAP_WS_WORDS - (HANDLER_PARAMS + 3)] =
      StartupPriority;
 CurrentPriorityPtr = &HandlerWorkspace[TRAP_WS_WORDS -
      (HANDLER_PARAMS + 3)];
}
}}}
{{{ enable desired traps
enable_traps (enable_trap_internal_channel |
       enable_trap_external_channel |
       enable_trap_timer      |
```

```
        enable_trap_run       |
        enable_trap_signal     |
        enable_trap_queue_empty,
        priority_low);
 }}}
 return TRUE;
}
}}}
{{{ void kernel_end (void)
void kernel_end (void)
{
 while (ActiveTasks > 0)
  {
   OS20SemWait (&TaskComplete);
   ActiveTasks--;
  }
 disable_traps (enable_trap_internal_channel |
        enable_trap_external_channel |
        enable_trap_timer      |
        enable_trap_run       |
        enable_trap_signal     |
        enable_trap_queue_empty,
        priority_low);
}
}}}
{{{ void task_end (void)
PRIVATE void task_end (void)
{
 __asm{ajw -4;};
 SemSignal (&TaskComplete);
 __asm {stopp;};
}
```

```
}}}
{{{ TASK* task_create (void (*Function)(), int StackSize, int
       Priority, int ParamWords, ...)
TASK* task_create (void (*Function)(), int StackSize, int Pri-
       ority, int ParamWords, ...)
{{{ stack layout at startup
/*************************************************************/
/*                                                           */
/*    2   first user param                                   */
/*    1   param 1 global static base                         */
/*    0   return address                                     */
/*   -1   entry point                                        */
/*   -2   undefined                                          */
/*   -3   undefined                                          */
/*   -4   undefined                                          */
/*   -5   undefined                                          */
/*   -6   priority                                           */
/*                                                           */
/*************************************************************/
}}}
{
 {{{ locals
 extern const volatile void* _params;
 TASK*     Task;
 unsigned int* Stack;
 int       i;
 int*      ParamList = (int*)((char*)&ParamWords + size-
       of(int));
 PARAMS*    GlobalParams = (PARAMS*)_params;
 }}}
 {{{ check the task list
 if (TaskCount >= MAX_TASKS)
```

```
 return NULL;
Task = TaskList[TaskCount];
}}}
{{{ malloc task stack
if (StackSize < DEFAULT_STACK_SIZE)
 StackSize = DEFAULT_STACK_SIZE;
Stack = (unsigned int*)malloc (StackSize);
if (Stack == NULL)
 return NULL;
}}}
{{{ initialise task structure
Task->t_id      = TaskCount++;
Task->t_state    = T_CREATED;
Task->t_stack_base  = Stack;
Task->t_stack_size  = StackSize;
Task->t_wptr     = (WORKSPACE)((BYTE*)Stack + (StackSize -
      (BYTES_PER_WORD * (ParamWords + 4))));
Task->t_priority   = Priority;
}}}
{{{ initialise positive workspace with params and return ad-
      dress
Task->t_wptr[0]  = (int)task_end;         /* return
      address */
Task->t_wptr[1]  = (int)GlobalParams->gsb;     /* global
      static base */
for (i = 0; i < ParamWords; i++)
 Task->t_wptr[i + 2] = ParamList[i];
}}}
{{{ initialise negative workspace ready to start
Task->t_wptr[PW_IPTR]   = (int)Function;
Task->t_wptr[PW_PRIORITY] = Priority;
}}}
{{{ register with Inquest if necessary
```

```
/*
#ifdef INQUEST
{
 int ChildIptr, ParentWptr;
 __asm
  {
   ldlp   0;
   st    ParentWptr;
   ld    (int)Task->t_wptr;
   ldnl   -1;
   st    ChildIptr;
  }
 IMSRTL_InformThreadBirth ((void*)ChildIptr,
              (void*)ParentWptr,
              (void*)Task->t_stack_base,
              (void*)(Task->t_stack_base + Task-
      >t_stack_size));
}
#endif */
}}}
{{{ start the task
Task->t_state = T_ACTIVE;
ActiveTasks++;
__asm
 {
  ld    (int)Task->t_wptr;
  ldpri;
  or;
  runp;   /* this will cause a trap and put the task on the
      correct queue */
 }
}}}
```

```
 return Task;
}
}}}
}}}
```

C. Appendix C

```
{{{ Title
*****************************************************************
 *                                                               *
 *   Module   : os20.c                                           *
 *   Purpose  : ST20 kernel (OS/20) scheduler trap handler       *
 *                                                               *
 *   Author   : Julian Wilson March 1996                         *
 *                                                               *
*****************************************************************/
}}}
{{{ includes
#include <stdlib.h>
#include <semaphor.h>
#include <rmclib.h>
#include ”os20.h”
#include ”library.h”
}}}
{{{ globals
TASK** CurrentTaskPtr;
}}}
{{{ statics
PRIVATE int*    CurrentPriorityPtr;
PRIVATE Semaphore TaskComplete;
PRIVATE int    ActiveTasks = 0;
PRIVATE TASK*   TaskList[MAX_TASKS];
PRIVATE int    TaskCount;
```

```
}}}
{{{ test globals
int HigherSchedules = 0;
int EqualSchedules  = 0;
int LowerSchedules  = 0;
int IdleSchedules   = 0;
int NewQueues     = 0;
int Idles       = 0;
int Resumes      = 0;
int TimeSlices    = 0;
}}}
{{{ functions
{{{ void* internal_memory_allocate (size_t Requested)
PRIVATE void* internal_memory_allocate (size_t Requested)
{
  PRIVATE unsigned int  Address  = MEMSTART;
  PRIVATE int       Available = RESERVED_INTERNAL;
  void*          Memory   = NULL;
  int   Claimed;
  if (Available > Requested)
   {
     Memory  = (void*)Address;
     Claimed  = (Requested + (BYTES_PER_WORD - 1)) & WORD_MASK;
     Address += Claimed;
     /*printf (”Claimed %d Bytes up to address %8X\n”, Claimed,
        Address - 1);*/
   }
  return Memory;
}
}}}
{{{ void  kernel_scheduler (int Areg, int Breg, int Creg)
{{{ tell compiler this is the trap handler
```

```
void kernel_scheduler (int Areg, int Breg, int Creg);
#pragma IMS_trap_handler (kernel_scheduler)
}}}
void kernel_scheduler (int Areg, int Breg, int Creg)
{
 extern const /*volatile */trap_state_t* _params;
 {{{ declare and place stacktics (on stack statics)
 int         QueueActive;
 QUEUE**       TaskQueue;
 trap_structure_t*  TrappedTask;
 TASK*        CurrentTask;
 int         CurrentPriority;
 #pragma IMS_place_at_workspace_offset (QueueActive,   9)
 #pragma IMS_place_at_workspace_offset (TaskQueue,    8)
 #pragma IMS_place_at_workspace_offset (TrappedTask,   7)
 #pragma IMS_place_at_workspace_offset (CurrentTask,   6)
 #pragma IMS_place_at_workspace_offset (CurrentPriority, 5)
 }}}
 if ((TrappedTask->status_word.word & enable_trap_queue_empty)
       != 0)
  {{{ find next task (on current or lower queue)
  {
   {{{ find active queue by looking at highest set bit in
       QueueActive
   __asm
    {
     ldab  0, QueueActive;
     norm;
     pop;
     pop;
     ldc   MAX_PRIORITY;
     rev;
```

```
  diff;
  st   CurrentPriority;
 }
}}}
if (CurrentPriority > 0)
 {
  QUEUE* Queue = TaskQueue[CurrentPriority];
  if (Queue->q_suspended)
   {{{ restore the preempted process
   {
    STATE* State = &Queue->q_pstate;
    /*Resumes++;*/
    {{{ CurrentTask = Queue->q_task;
    __asm
     {
      ld  Queue->q_task;
      st  CurrentTask;
     }
    }}}
    Queue->q_suspended = FALSE;
    if (Queue->q_front == (TASK*)QUEUE_EMPTY)
     {{{ QueueActive ^= 1 << CurrentPriority;
     __asm
      {
       ldab  CurrentPriority, 1;
       shl;
       ld   QueueActive;
       xor;
       st   QueueActive;
      }
     }}}
    State->s_cpu_state.c_status &=
```

```
  CLEAR_TRAP_REASON_MASK;
  __asm
   {
    ld  State;
    restart;
   }
 }
 }}}
else
 {{{ remove and run only the front process
 {
  TASK* Front = Queue->q_front;
  /*NewQueues++;*/
  {{{ CurrentTask = Front;
  __asm
   {
    ld  Front;
    st  CurrentTask;
   }
  }}}
  Queue->q_front  = Front->t_link;
  if (Queue->q_front == (TASK*)QUEUE_EMPTY)
   {{{ QueueActive ^= 1 << CurrentPriority;
   __asm
    {
     ldab  CurrentPriority, 1;
     shl;
     ld   QueueActive;
     xor;
     st   QueueActive;
    }
   }}}
```

```
        {{{ copy Front details into TrappedTask structure
        TrappedTask->status_word.bits.process_valid = TRUE;
        TrappedTask->wptr              = (int*)Front-
        >t_wptr;
        TrappedTask->iptr.address          =
        (void*)Front->t_iptr;
        }}}
        __asm {tret;}
      }
      }}}
    }
   /*Idles++;*/
   {{{ disable idle trap
   TrappedTask->enables_word &= (~enable_trap_queue_empty);
   }}}
  }
 }}}
else if ((TrappedTask->status_word.word &
       enable_trap_timeslice) != 0)
 {{{ put the process onto the back of the current queue
 {
  int TaskPriority = CurrentTask->t_priority;
  if (TaskPriority == CurrentPriority)
   {{{ timeslice the task
   {
    QUEUE* Queue = TaskQueue[CurrentPriority];
    /*TimeSlices++;*/
    if (Queue->q_front != (TASK*)QUEUE_EMPTY)
     {
      TASK* OldTask = CurrentTask;
      TASK* NewTask;
      {{{ put current task at back of queue
```

```
    OldTask->t_wptr    = (WORKSPACE)TrappedTask->wptr;
    OldTask->t_iptr    = (void*)TrappedTask-
    >iptr.address;
    OldTask->t_link    = (TASK*)QUEUE_EMPTY;
    Queue->q_back->t_link = OldTask;
    Queue->q_back     = OldTask;
    }}}
    {{{ access front of queue and write to trapped task
    structure
    NewTask          = Queue->q_front;
    TrappedTask->wptr     = (int*)NewTask->t_wptr;
    TrappedTask->iptr.address = NewTask->t_iptr;
    Queue->q_front       = NewTask->t_link;
    {{{ CurrentTask = NewTask;
    __asm
     {
      ld  NewTask;
      st  CurrentTask;
     }
    }}}
    }}}
   }
 }
 }}}
else if (TaskPriority > CurrentPriority)
 {{{ task is raising its priority - just reset current
    priority
 {{{ CurrentPriority = TaskPriority;
 __asm
  {
   ld  TaskPriority;
   st  CurrentPriority;
```

```
    }
   }}}
   }}}
  else
   {{{ task is lowering its priority - put on back of
       appropriate queue
   {
    QUEUE* Queue = TaskQueue[TaskPriority];
    TASK*  OldTask = CurrentTask;
    OldTask->t_wptr = (WORKSPACE)TrappedTask->wptr;
    OldTask->t_iptr = (void*)TrappedTask->iptr.address;
    OldTask->t_link = (TASK*)QUEUE_EMPTY;
    if (Queue->q_front != (TASK*)QUEUE_EMPTY)
     Queue->q_back->t_link = OldTask;
    else
     Queue->q_front = OldTask;
    Queue->q_back = OldTask;
    {{{ QueueActive |= 1 << TaskPriority;
    __asm
     {
      ldab  TaskPriority, 1;
      shl;
      ld   QueueActive;
      or;
      st   QueueActive;
     }
    }}}
    TrappedTask->status_word.bits.process_valid = FALSE;
   }
   }}}
  __asm {tret;}
 }
```

```
 }}}
else
 {{{ schedule the process onto a queue
 {
  WORKSPACE  NewWptr   = (WORKSPACE)((int)_params->wdesc &
      (~PRIORITY_MASK));
  TASK*    NewTask   = (TASK*)NewWptr[PW_TASK];
  int     NewPriority = NewTask->t_priority;
  if (NewPriority > CurrentPriority)
   {{{ replace current queue if there is one
   {
    /*HigherSchedules++;*/
    if (CurrentPriority < 0)
     {{{ cpu was idle, set trap bits to valid
     {
      TrappedTask->enables_word          |=
      enable_trap_queue_empty;
      TrappedTask->status_word.bits.process_valid = TRUE;
     }
     }}}
    else
     {{{ save current state
     {
      QUEUE*  CurrentQueue = TaskQueue[CurrentPriority];
      STATE*  State = &(CurrentQueue->q_pstate);
      __asm
       {
        #if 0
        ldabc   MOVE2D_SAVE_MASK, priority_low, State;
        stshadow;     /* save move 2D information */
        #endif
        ldabc   ABC_SIZE_IN_BYTES, &State->s_cpu_stack,
```

```
  &Areg;
    move;       /* copy A, B and C from Params */
    ldabc   SCHEDULER_TRAP, &State->s_cpu_state,
  priority_low;
    sttrapped;    /* copy trapped task state */
   }
  CurrentQueue->q_suspended = TRUE;
  CurrentQueue->q_task    = CurrentTask;
  {{{ QueueActive |= 1 << CurrentPriority;
  __asm
   {
    ldab  CurrentPriority, 1;
    shl;
    ld   QueueActive;
    or;
    st   QueueActive;
   }
  }}}
 }
 }}}
{{{ CurrentPriority = NewPriority;
__asm
 {
  ld  NewPriority;
  st  CurrentPriority;
 }
}}}
{{{ CurrentTask = NewTask;
__asm
 {
  ld  NewTask;
  st  CurrentTask;
```

```
    }
   }}}
   {{{ reset workspace and iptr in TrappedTask structure
   TrappedTask->wptr     = (int*)NewWptr;
   TrappedTask->iptr.address = (void*)NewWptr[PW_IPTR];
   }}}
   __asm {tret;}
  }
  }}}
 else
  {{{ add to lower priority queue
  {
   QUEUE* Queue = TaskQueue[NewPriority];
   /*LowerSchedules++;*/
   NewTask->t_wptr = NewWptr;
   NewTask->t_iptr = (void*)NewWptr[PW_IPTR];
   NewTask->t_link = (TASK*)QUEUE_EMPTY;
   if (Queue->q_front != (TASK*)QUEUE_EMPTY)
    Queue->q_back->t_link = NewTask;
   else
    Queue->q_front = NewTask;
   Queue->q_back = NewTask;
   {{{ QueueActive |= 1 << NewPriority;
   __asm
    {
     ldab  NewPriority, 1;
     shl;
     ld   QueueActive;
     or;
     st   QueueActive;
    }
   }}}
```

```
      {{{ restore registers for trapped task
      __asm
       {
        ldabc  Areg, Breg, Creg;
       }
      }}}
     }
     }}}
   }
   }}}
}
}}}
{{{ BOOL kernel_start (int PriorityLevels)
BOOL kernel_start (int PriorityLevels)
{
 TASK*  Task;
 QUEUE* TaskQueue;
 QUEUE** TaskQueuePtr;
 int   StartupPriority = PriorityLevels - 1;
 int   QueueActive;
 {{{ malloc space for and intialise queues and lists
 {
  int  i;
  TaskQueuePtr = internal_memory_allocate (sizeof(QUEUE*) *
       PriorityLevels);
  TaskQueue   = internal_memory_allocate (sizeof(QUEUE) *
       PriorityLevels);
  for (i = 0; i < PriorityLevels; i++)
   {
    TaskQueuePtr[i]      = &TaskQueue[i];
    TaskQueue[i].q_front   = (TASK*)QUEUE_EMPTY;
    TaskQueue[i].q_suspended = FALSE;
```

```
  }
 Task = internal_memory_allocate (sizeof(TASK) * MAX_TASKS);
 for (i = 0; i < MAX_TASKS; i++)
  TaskList[i] = &Task[i];
}
}}}
{{{ initialise global state
TaskCount  = 0;
QueueActive = 0;
SemInit (&TaskComplete, 0);
}}}
{{{ create kernel task
Task       = TaskList[TaskCount];
Task->t_id    = TaskCount++;
Task->t_state   = T_ACTIVE;
Task->t_priority = StartupPriority;
Task->t_link   = (TASK*)QUEUE_EMPTY;
TaskQueue[StartupPriority].q_front  = (TASK*)QUEUE_EMPTY;
TaskQueue[StartupPriority].q_back  = (TASK*)QUEUE_EMPTY;
}}}
{{{ create trap handler environment and install it
{
 int* HandlerWorkspace;
 HandlerWorkspace = internal_memory_allocate (TRAP_WS_SIZE);
 if (HandlerWorkspace == NULL)
  return FALSE;
 install_trap_handler (trap_group_scheduler,
            priority_low,
            (enables_word_t)EVERYTHING_DISABLED,
            default_status_register,
            HandlerWorkspace,
            TRAP_WS_SIZE,
```

```
            kernel_scheduler);
  HandlerWorkspace[TRAP_WS_WORDS - (HANDLER_PARAMS + 1)] =
       QueueActive;
  HandlerWorkspace[TRAP_WS_WORDS - (HANDLER_PARAMS + 2)] =
       (int)TaskQueuePtr;
  HandlerWorkspace[TRAP_WS_WORDS - (HANDLER_PARAMS + 3)] =
       (int)LOW_TRAPPED_TASK;
  HandlerWorkspace[TRAP_WS_WORDS - (HANDLER_PARAMS + 4)] =
       (int)Task;
  HandlerWorkspace[TRAP_WS_WORDS - (HANDLER_PARAMS + 5)] =
       StartupPriority;
  CurrentTaskPtr   = (TASK**)&HandlerWorkspace[TRAP_WS_WORDS
       - (HANDLER_PARAMS + 4)];
  CurrentPriorityPtr = &HandlerWorkspace[TRAP_WS_WORDS -
       (HANDLER_PARAMS + 5)];
 }
 }}}
 {{{ enable desired traps
 enable_traps (enable_trap_internal_channel |
        enable_trap_external_channel |
        enable_trap_timer       |
        enable_trap_run        |
        enable_trap_signal      |
        /*enable_trap_timeslice     |*/
        enable_trap_queue_empty,
        priority_low);
 }}}
 return TRUE;
}
}}}
{{{ void  kernel_end (void)
void kernel_end (void)
{
```

```
 while (ActiveTasks > 0)
  {
   OS20SemWait (&TaskComplete);
   ActiveTasks--;
  }
 disable_traps (enable_trap_internal_channel |
        enable_trap_external_channel |
        enable_trap_timer       |
        enable_trap_run        |
        enable_trap_signal      |
        enable_trap_timeslice    |
        enable_trap_queue_empty,
        priority_low);
}
}}}
{{{ void  task_end (void)
PRIVATE void task_end (void)
{
 __asm{ajw -4;};
 SemSignal (&TaskComplete);
 __asm {stopp;};
}
}}}
{{{ TASK* task_create (void (*Function)(), int StackSize, int
       Priority, int ParamWords, ...)
TASK* task_create (void (*Function)(), int StackSize, int Pri-
       ority, int ParamWords, ...)
{{{ stack layout at startup
/*************************************************************/
/*                                                           */
/*    2   first user param                                   */
/*    1   param 1 global static base                         */
```

```
/*     0   return address                                          */
/*    -1   entry point                                            */
/*    -2   undefined                                              */
/*    -3   undefined                                              */
/*    -4   undefined                                              */
/*    -5   undefined                                              */
/*    -6   task descriptor                                        */
/*                                                                */
/******************************************************************/
}}}
{
 {{{ locals
 extern const volatile void* _params;
 TASK*      Task;
 unsigned int* Stack;
 int       i;
 int*       ParamList = (int*)((char*)&ParamWords + size-
       of(int));
 PARAMS*     GlobalParams = (PARAMS*)_params;
 }}}
 {{{ check the task list
 if (TaskCount >= MAX_TASKS)
  return NULL;
 Task = TaskList[TaskCount];
 }}}
 {{{ malloc task stack
 if (StackSize < DEFAULT_STACK_SIZE)
  StackSize = DEFAULT_STACK_SIZE;
 Stack = (unsigned int*)malloc (StackSize);
 if (Stack == NULL)
  return NULL;
 }}}
```

```
{{{ initialise task structure
Task->t_id      = TaskCount++;
Task->t_state    = T_CREATED;
Task->t_stack_base  = Stack;
Task->t_stack_size  = StackSize;
Task->t_wptr     = (WORKSPACE)((BYTE*)Stack + (StackSize -
      (BYTES_PER_WORD * (ParamWords + 4))));
Task->t_priority   = Priority;
}}}
{{{ initialise positive workspace with params and return ad-
      dress
Task->t_wptr[0]  = (int)task_end;          /* return
      address */
Task->t_wptr[1]  = (int)GlobalParams->gsb;     /* global
      static base */
for (i = 0; i < ParamWords; i++)
 Task->t_wptr[i + 2] = ParamList[i];
}}}
{{{ initialise negative workspace ready to start
Task->t_wptr[PW_IPTR] = (int)Function;
Task->t_wptr[PW_TASK] = (int)Task;
}}}
{{{ register with Inquest if necessary
/*
#ifdef INQUEST
{
 int ChildIptr, ParentWptr;
 __asm
  {
   ldlp   0;
   st    ParentWptr;
   ld    (int)Task->t_wptr;
   ldnl   -1;
```

```
    st    ChildIptr;
   }
  IMSRTL_InformThreadBirth ((void*)ChildIptr,
              (void*)ParentWptr,
              (void*)Task->t_stack_base,
              (void*)(Task->t_stack_base + Task-
      >t_stack_size));
 }
 #endif */
 }}}
 {{{ start the task
 Task->t_state = T_ACTIVE;
 ActiveTasks++;
 __asm
  {
   ld    (int)Task->t_wptr;
   ldpri;
   or;
   runp;   /* this will cause a trap and put the task on the
       correct queue */
  }
 }}}
 return Task;
}
}}}
{{{ TASK* task_id (void)
TASK* task_id (void)
{
 return *CurrentTaskPtr;
}
}}}
{{{ int  task_priority (int Delta)
```

```
int task_priority (int Delta)
{
 int  OldPriority, NewPriority;
 OldPriority = *CurrentPriorityPtr;
 NewPriority = OldPriority + Delta;
 if (NewPriority > MAX_USER_PRIORITY)
  NewPriority = MAX_USER_PRIORITY;
 else if (NewPriority < MIN_USER_PRIORITY)
  NewPriority = MIN_USER_PRIORITY;
 if (NewPriority != OldPriority)
  {
   (*CurrentTaskPtr)->t_priority = NewPriority;
   __asm {timeslice;}
  }
 return OldPriority;
}
}}}
}}}
```