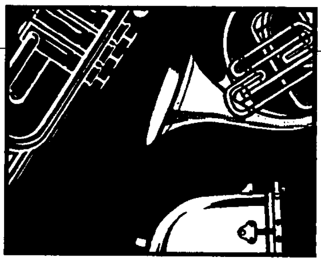

Specifying a Real-Time Kernel

J. Michael Spivey, Tektronix

This case study of an embedded real-time kernel shows that mathematical techniques have an important role to play in documenting systems and avoiding design flaws.

Embedded systems are commonly built around a small operating-system kernel that provides process-scheduling and interrupt-handling facilities. This article reports on a case study I made, using the Z notation,[1,2] a mathematical specification language, to specify the kernel for a diagnostic X-ray machine.

Beginning with the documentation and source code of an existing implementation, I constructed a mathematical model, expressed in Z, of the states that the kernel could occupy and the events that could take it from one state to another. My goal was a precise specification that could be used as a basis for a new implementation on different hardware.

This case study in specification had a surprising by-product, because in studying one of the kernel's operations, I discovered that it could sometimes lead to deadlock: The kernel would disable interrupts and enter a tight loop, vainly searching for a process ready to run.

This flaw in the kernel's design was reflected directly in a mathematical property of its specification, demonstrating how formal techniques can help avoid design errors. This help should be especially welcome in embedded systems, which are notoriously difficult to test effectively.

A conversation with the kernel designer later revealed that, for two reasons, the design error did not in fact endanger patients using the X-ray machine. First, the actual control software happened to avoid the circumstances that could lead to deadlock. Second, there was a hardware timeout that protected against hardware or software failure in the machine. Nevertheless, the error seriously affected the X-ray machine's robustness and reliability because later enhancements to the controlling software might reveal the problem with deadlock that had been hidden before.

I have simplified the specification presented in this article by making less use of

0740-7459/90/0900/0021/$01.00 © 1990 IEEE

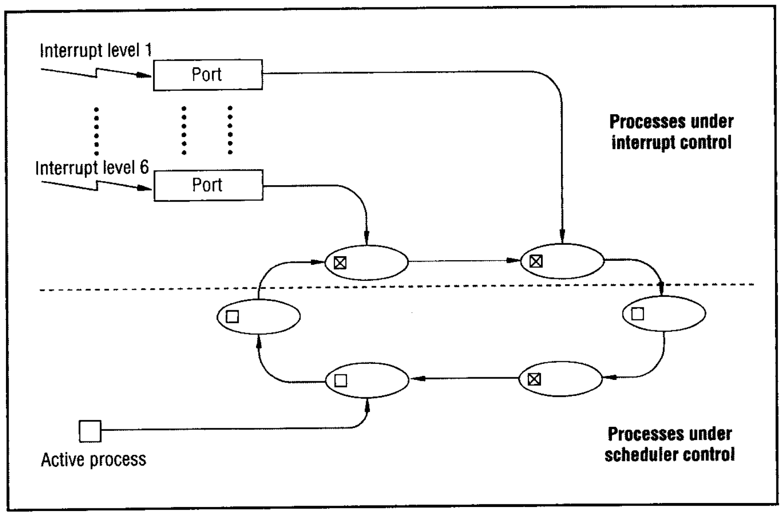
Interrupt level 1
Port
Interrupt level 6
Port
Processes under interrupt control
Active process
Processes under scheduler control

Figure 1. Kernel data structures: Processes, shown as ovals, are linked in a ring. The processes above the horizontal dotted line are the interrupt handlers, and those below it are background processes. The ready flag for each process is shown as a small square.

the schema calculus, a way of structuring Z specifications. This has made the specification a little longer and more repetitive but perhaps a little easier to follow without knowledge of Z.

About the kernel

The kernel supports both background processes and interrupt handlers. There may be several background processes, and one may be marked as current. This process runs whenever no interrupts are active, and it remains current until it explicitly releases the processor; the kernel may then select another process to be current. Each background process has a ready flag, and the kernel chooses the new current process from among those with a ready flag set to true.

When interrupts are active, the kernel chooses the most urgent according to a numerical priority, and the interrupt handler for that priority runs. An interrupt may become active if it has a higher priority than those already active, and it becomes inactive again when its handler signals that it has finished. A background process may become an interrupt handler by registering itself as the handler for a certain priority.

Documentation

Figures 1 and 2 are diagrams from the existing kernel documentation, typical of the ones used to describe kernels like this. Figure 1 shows the kernel data structures. Figure 2 shows the states that a single process may occupy and the possible transitions between them, caused either by a kernel call from the process itself or by some other event.

Although diagrams like these are fairly common, they convey little information that a programmer can use in developing programs to run under the kernel, and they contain much that is irrelevant or distracting. For example, Figure 1 shows that interrupt handlers are linked into the ring with the background processes and that they have a ready flag. These are facts that a programmer cannot detect.

The fact that a scheduling ring exists at all, so background processes run in a fixed order, is one that programmers using the kernel should *not* exploit if their programs are to be robust. Also, because the diagram shows just one possible state of the kernel, it can be misleading. For example, the interrupt handlers are shown as occupying consecutive places in the ring, although this need not be true.

In a way, Figure 2 is a partial specification of the kernel as a set of finite-state machines, one for each process. However, it gives no explicit information about the interactions between processes — the very thing the kernel is required to manage. Also, it fails to show several possible states of a process. For example, the current background process may not be ready if it has set its own ready flag to false, but the state "current but not ready" is not shown in the diagram. Correcting this defect would require adding two more states and seven more transitions. This highlights another deficiency of state diagrams like this: Their size tends to grow exponentially as system complexity increases.

Kernel state

Like most Z specifications, the kernel model begins with a description of the *state space*: the collection of variables that determine what state the kernel is in and the invariant relationships that always hold between these variables' values.

In this article, I describe the kernel state space in several parts, corresponding to the background processes, the interrupt handlers, and the state of the processor on which the kernel runs. Each piece is described by a schema, the basic unit of specification in Z. (The box on p. 25 describes the Z notation used in this article.) I obtain the state space of the whole kernel by putting together these pieces and adding more invariants, among them the static policy for allocating the processor to a background process or interrupt handler.

Processes are named in the kernel by *process identifiers*. In the implemented kernel, these are the addresses of process-control blocks, but this detail is irrelevant to a programmer using the kernel, so I introduce them as a basic type PID:

$[PID]$

This declaration introduces PID as the name of a set, without giving any information about its members. From the specification's point of view, the members are simply atomic objects.

For convenience, I introduced also the fictitious process identifier none, which is not the name of any genuine process. When the processor is idle, the current process is none. The set PID_1 contains all process identifiers except none:

$$\begin{array}{|l}
none : PID \\
PID_1 : \mathbb{P}\, PID \\
\hline
PID_1 = PID \setminus \{none\}
\end{array}$$

The part of the kernel state concerned with background processes is described by this schema:

$$\begin{array}{|l}
\text{—}Scheduler\text{————} \\
background : \mathbb{P}\, PID_1 \\
ready : \mathbb{P}\, PID_1 \\
current : PID \\
\hline
ready \subseteq background \\
current \in background \cup \{none\} \\
\hline
\end{array}$$

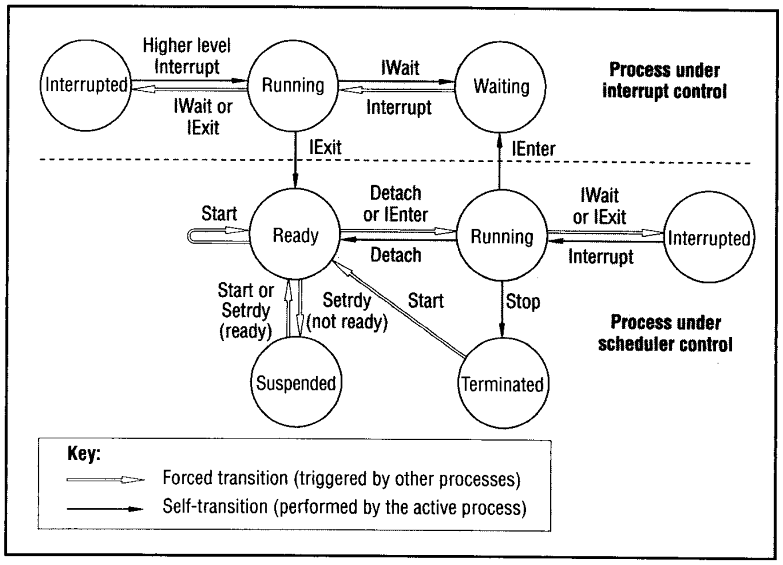

Figure 2. The states that a single process may occupy and the possible transitions between them, caused either by a kernel call from the process itself (thin arrows) or by some other event (thick arrows).

Like all schemas, this one declares some typed variables and states a relationship between them. Above the horizontal dividing line, in the *declaration* part, the three variables background, ready, and current are declared:

• Background is the set of processes under control of the scheduler.

• Ready is the set of processes that may be selected for execution when the processor becomes free.

• Current is the process selected for execution in the background.

Below the line, in the predicate part, the schema states two relationships that always hold. The set ready is always a subset of background, and current is either a member of background or the fictitious process none. This schema lets the current process be not ready, because no relationship between current and ready is specified.

This schema does not reflect the full significance of the set ready, but its real meaning will be shown later in the Select operation, where it forms the pool from which a new current process is chosen.

Interrupts are identified by their priority levels, which are small positive integers. The finite set ILEVEL includes all the priorities:

$ILEVEL : \mathbb{F}\,\mathbb{N}$

$0 \notin ILEVEL$

Zero is not one of the interrupt priority levels, but it is the priority associated with background processes.

The state space for the interrupt-handling part of the kernel is described like this:

IntHandler

$handler : ILEVEL ⤔ PID_1$

$enabled, active : \mathbb{P}\ ILEVEL$

$enabled \cup active \subseteq \mathrm{dom}\ handler$

This schema declares the three variables handler, enabled, and active:

• Handler is a function that associates certain priority levels with processes, the interrupt handlers for those priorities.

• Enabled is the set of priority levels that are enabled, so an interrupt can happen at that priority.

• Active is the set of priority levels for which an interrupt is being handled.

The predicate part of this schema says that each priority level that is either enabled or active must be associated with a handler. An interrupt may be active without being enabled if, for example, it has been disabled by the handler itself since becoming active.

The declaration

$handler : ILEVEL ⤔ PID_1$

declares handler to be a partial injection. It is partial in that not every priority level need be associated with an interrupt handler, and it is an injection in that no two distinct priority levels may share the same handler.

Information like this — that interrupts may be active without being enabled and that different priority levels may not share a handler — is vital to understanding how the kernel works. It is especially valuable because it is static information about the states the kernel may occupy, rather than about what happens as the system moves from state to state. Such static information is often absent from the text of the program, which consists mostly of dynamic, executable code.

The state space of the whole kernel combines the background and interrupt parts:

Kernel

Scheduler

IntHandler

$background \cap \mathrm{ran}\ handler = \varnothing$

The declarations Scheduler and IntHandler in this schema implicitly include above the line all the declarations from those schemas and implicitly include below the line all their invariants. So the schema Kernel has six variables: background, ready, and current from Scheduler, and handler, enabled, and active from IntHandler.

I have added the additional invariant that no process can be both a background process and an interrupt handler at the same time.

The main job of the kernel is to control which process the processor runs and at what priority, so I made the running process and the processor priority part of the state of the system. Here is a schema that declares them:

CPU

$running : PID$

$priority : ILEVEL \cup \{0\}$

This schema has an empty predicate part, so it places no restriction on the values of its variables, except that each must be a member of its type. The variable running takes the value none when no process is running.

Of course, there are many other parts of the processor state, including the contents of registers and memory, condition codes, and so on, but they are irrelevant in this context.

Because the kernel always uses the same scheduling policy to select the running process and the CPU priority, this policy is another invariant of the system. It is stated in the schema State, which combines the kernel and CPU parts of the system state:

$$
\begin{array}{|l}
\textit{State} \\
\hline
\textit{Kernel} \\
\textit{CPU} \\
\hline
\textit{priority} = \textit{max}(\textit{active} \cup \{0\}) \\
\textit{priority} = 0 \Rightarrow \textit{running} = \textit{current} \\
\textit{priority} > 0 \Rightarrow \textit{running} = \textit{handler}(\textit{priority}) \\
\hline
\end{array}
$$

If any interrupts are active, the processor priority is the highest priority of an active interrupt, and the processor runs the interrupt handler for that priority. Otherwise, the processor runs the current background process at priority zero.

The invariant part of this schema uniquely determines priority and running in terms of active, current, and handler, three variables of the schema Kernel. I will exploit this fact when I describe events that change the system state.

With the description of the kernel's and processor's state space complete, the next step is to look at the operations and events that can change the state.

Background processing

Some kernel operations affect only the background-processing part of the state space. They start background processes, set and clear their ready flags, let them release the processor temporarily or permanently, and select a new process to run when the processor is idle.

A process enters the control of the scheduler through the operation Start, described by this schema:

$$
\begin{array}{|l}
\textit{Start} \\
\hline
\Delta \textit{State} \\
\textit{p?} : \textit{PID}_1 \\
\hline
\textit{p?} \notin \operatorname{ran} \textit{handler} \\
\\
\textit{background}' = \textit{background} \cup \{\textit{p?}\} \\
\textit{ready}' = \textit{ready} \cup \{\textit{p?}\} \\
\textit{current}' = \textit{current} \\
\theta \textit{IntHandler}' = \theta \textit{IntHandler} \\
\hline
\end{array}
$$

Like all schemas describing operations, this one includes the declaration ΔState, which implicitly declares two copies of each variable in the state space State, one with a prime (′) and one without. Variables like background and ready without a prime refer to the system state before the operation has happened, and variables like background′ and ready′ with a prime refer to the state afterward.

The declaration ΔState also implicitly constrains these variables to obey the invariant relationships I documented in defining the schema State — including the scheduling policy — so they hold both before and after the operation.

In addition to these state variables, the Start operation has an input *p?*, the identifier of the process to be started. By convention, inputs to operations are given names that end in a ?.

The predicate part of an operation schema lists the precondition that must be true when the operation is invoked and postcondition that must be true afterward. In this case, the precondition is explicitly stated: that the process *p?* being started must not be an interrupt handler (because that would violate the invariant that background processes are disjoint from interrupt handlers).

Some kernel operations affect only the background-processing part of the state space. They start background processes, set and clear their ready flags, let them release the processor, and select a new process to run when the processor is idle.

In general, an operation's precondition is that a final state exists that satisfies the predicates written in the schema. Part of the precondition may be implicit in the predicates that relate the initial and final states. If an operation is specified by the schema Op, its precondition can be calculated as

$$\exists\, \textit{State} \bullet \textit{Op}$$

If the precondition is true, the specification requires that the operation should terminate in a state satisfying the postcondition. On the other hand, if the precondition is false when the operation is invoked, the specification says nothing about what happens. The operation may fail to terminate, or the kernel may stop working completely.

For the Start operation, the postcondition says that the new process is added to the set of background processes and marked as ready to run. The new process does not start to run immediately, because current is unchanged; instead, the processor continues to run the same process as before.

The final equation in the postcondition

$$\theta \textit{IntHandler}' = \theta \textit{IntHandler}$$

means that the part of the state described by the schema IntHandler is the same after the operation as before it.

The equations in this schema determine the final values of the six variables in the kernel state space in terms of their initial values and the input *p?*, but they say nothing about the final values of the CPU variables running and priority. These are determined by the requirement, implicit in the declaration ΔState, that the scheduling policy be obeyed after the operation has finished. Because the values of active, handler, and current do not change in the operation, neither does the CPU state.

The current background process may release the processor by calling the Detach operation, specified like this:

$$
\begin{array}{|l}
\textit{Detach} \\
\hline
\Delta \textit{State} \\
\hline
\textit{running} \in \textit{background} \\
\\
\textit{background}' = \textit{background} \\
\textit{ready}' = \textit{ready} \\
\textit{current}' = \textit{none} \\
\theta \textit{IntHandler}' = \theta \textit{IntHandler} \\
\hline
\end{array}
$$

Again, this operation is described using ΔState in terms of the values of state variables before and after the operation has happened. The precondition is that the processor is running a background process. The only change specified in the postcondition is that the current process changes to none, meaning that the processor is now idle. The next event will be either an interrupt or the selection of a new background process to run.

After a call to Detach — and after other operations I describe later — current has

Summary of Z notation

Z notation is based on typed set theory and first-order logic. Z provides a construct, called a schema, to describe a specification's state space and operations. A schema groups variable declarations with a list of predicates that constrain the variables' possible values.

In Z, the schema *X* is defined by the form

```
┌─X──────────────────────────────
│ declarations
├──────────────
│ predicates
└────────────────────────────────
```

Global functions and constants are defined by the form

```
│ declarations
├──────────────
│ predicates
```

The declaration gives the type of the function or constant, while the predicate gives its value.

Here, I define only the Z symbols used in this article:

Sets:

$S : \mathbb{P}\,X$	*S* is declared as a set of *X*'s.
$x \in S$	*x* is a member of *S*.
$x \notin S$	*x* is not a member of *S*.
$S \subseteq T$	*S* is a subset of *T*: Every member of *S* is also in *T*.
$S \cup T$	The union of *S* and *T*: It contains every member of *S* or *T* or both.
$S \cap T$	The intersection of *S* and *T*: It contains every member of both *S* and *T*.
$S \setminus T$	The difference of *S* and *T*: It contains every member of *S* except those also in *T*.
$\varnothing$	Empty set: It contains no members.
$\{x\}$	Singleton set: It contains just *x*.
$\mathbb{N}$	The set of natural numbers 0, 1, 2, ...
$S : \mathbb{F}\,X$	*S* is declared as a finite set of *X*'s.
$max(S)$	The maximum of the nonempty set of numbers *S*.

Functions:

$f : X \rightarrowtail\!\!\!\!\!\!+\ Y$	*f* is declared as a partial injection from *X* to *Y* (described in the handler definition on p. 23).
dom *f*	The domain of *f*: the set of values *x* for which *f*(*x*) is defined.
ran *f*	The range of *f*: the set of values taken by *f*(*x*) as *x* varies over the domain of *f*.
$f \oplus \{x \mapsto y\}$	A function that agrees with *f* except that *x* is mapped to *y*.
$\{x\} \lhd f$	A function like *f*, except that *x* is removed from its domain.

Logic:

$P \wedge Q$	*P* and *Q*: It is true if both *P* and *Q* are true.
$P \Rightarrow Q$	*P* implies *Q*: It is true if either *Q* is true or *P* is false.
$\theta S' = \theta S$	No components of schema *S* change in an operation.

value none, indicating that no background process has been selected for execution. If no interrupts are active, the processor is idle, and the Select operation may happen spontaneously. It is specified like this:

```
┌─Select─────────────────────────
│ Δ State
├──────────────
│ running = none
│
│ background' = background
│ ready' = ready
│ current' ∈ ready
│ θIntHandler' = θIntHandler
└────────────────────────────────
```

Rather than a part of the interface between the kernel and an application, Select is an internal operation of the kernel that can happen whenever its precondition is true. The precondition is

$running = none \wedge ready \neq \varnothing$

The processor must be idle, and at least one background process must be ready to run. The first part of this precondition is stated explicitly, and the second part is implicit in the predicate

$current' \in ready$

The new value of current is selected from ready, but the specification does not say how the choice is made — it is nondeterministic. This nondeterminism lets the specification say exactly what programmers may rely on the kernel to do: There is no guarantee that processes will be scheduled in a particular order.

In fact, the nondeterminism is a natural consequence of the abstract view I have taken in the specification. Although the program that implements this specification is deterministic — if started with the ring of processes in a certain state, it will always select the same process — it appears to be nondeterministic if you pay attention only to the *set* of processes that are ready, as I have done in the specification.

However the kernel selects the new current process, the specification says that it starts to run, because of the static scheduling policy, which determines that after the operation, running is current and priority is zero.

A background process may terminate itself using the Stop operation:

```
┌─Stop───────────────────────────
│ ΔState
├──────────────
│ running ∈ background
│
│ background' = background \ {current}
│ ready' = ready \ {current}
│ current' = none
│ θIntHandler' = θIntHandler
└────────────────────────────────
```

For this operation to be permissible, the processor must be running a background process. This process is removed from background and ready, and the current process becomes none, so the next action will be to select another process.

A final operation, SetReady, sets or clears a process's ready flag. It has two inputs, the process identifier and a flag, which takes one of the values set or clear:

$FLAG ::= set \mid clear$

The SetReady operation is:

```
┌─SetReady───────────────────────
│ ΔState
│ p?: PID
│ flag?: FLAG
├──────────────
│ p? ∈ background
│
│ flag? = set ⇒ ready' = ready ∪ {p?}
│ flag? = clear ⇒ ready' = ready \ {p?}
│ background' = background
│ current' = current
│ θIntHandler' = θIntHandler
└────────────────────────────────
```

The precondition is that *p*? is a background process; according to the value of flag?, it is either inserted in ready or removed from it. The scheduling parameters do not change, so there is no change in the running process.

Interrupt handling

Other operations affect the kernel's interrupt-handling part. A background process may register itself as the handler for a certain priority level by calling the operation IEnter:

$$\begin{array}{|l}
\textit{IEnter} \\
\Delta State \\
i?: ILEVEL \\
\hline
running \in background \\
\\
background' = background \setminus \{current\} \\
ready' = ready \setminus \{current\} \\
current' = none \\
handler' = handler \oplus \{i? \mapsto current\} \\
enabled' = enabled \cup \{i?\} \\
active' = active \\
\hline
\end{array}$$

This operation may be called only by a background process. The operation removes the calling process from background and ready, and the new value of current is none, just as in the Stop operation. Also, the calling process becomes an interrupt handler for the priority level $i?$, given as an input to the operation, and that priority level becomes enabled.

The expression

$$handler \oplus \{i? \mapsto current\}$$

denotes a function identical to handler, except that $i?$ is mapped to current. This function is an injection, because current, a background process, cannot already be the handler for any other priority level. The new handler supersedes any existing handler for priority $i?$, which can never run again unless restarted with the Start operation.

Once a process has registered itself as an interrupt handler, the scheduler chooses a new process to run in the background, and the new interrupt handler waits for an interrupt to happen:

$$\begin{array}{|l}
\textit{Interrupt} \\
\Delta State \\
i?: ILEVEL \\
\hline
i? \in enabled \;\wedge\; i? > priority \\
\\
\theta Scheduler' = \theta Scheduler \\
handler' = handler \\
enabled' = enabled \\
active' = active \cup \{i?\} \\
\hline
\end{array}$$

The processor hardware ensures that interrupts happen only when they are enabled and have a priority greater than the processor priority. If these conditions are satisfied, the interrupt can happen and the kernel then adds the interrupt to active.

The scheduling policy ensures that the associated interrupt handler starts to run.

The processor hardware ensures that interrupts happen only when they are enabled and have a priority greater than the processor priority. If these are true, the interrupt can happen and the kernel then adds the interrupt to active.

In this calculation of the new processor priority, each step is justified by the comment in brackets:

$$\begin{array}{ll}
 & priority' \\
= & [\text{scheduling policy}] \\
 & max(active' \cup \{0\}) \\
= & [\text{postcondition}] \\
 & max((active \cup \{i?\}) \cup \{0\}) \\
= & [\cup \text{ assoc. and comm.}] \\
 & max((active \cup \{0\}) \cup \{i?\}) \\
= & [\text{max dist. over } \cup] \\
 & max\{max(active \cup \{0\}), i?\} \\
= & [\text{scheduling policy}] \\
 & max\{priority, i?\} \\
= & [i? > \text{priority}] \\
 & i?
\end{array}$$

So priority′ = $i? > 0$ and the other part of the scheduling policy ensures that running′ equals handler($i?$).

After the interrupt handler has finished the processing associated with the interrupt, it calls the kernel operation IWait and suspends itself until another interrupt arrives. IWait is specified as

$$\begin{array}{|l}
\textit{IWait} \\
\Delta State \\
\hline
priority > 0 \\
\\
\theta Scheduler' = \theta Scheduler \\
handler' = handler \\
enabled' = enabled \\
active' = active \setminus \{priority\} \\
\hline
\end{array}$$

The precondition priority > 0 means that the processor must be running an interrupt handler. The current priority level is removed from active, and as for the Interrupt operation, the scheduling policy determines what happens next. If any other interrupts are active, the processor returns to the interrupt handler with the next highest priority. Otherwise, it returns to the current background process.

Another kernel operation, IExit, lets an interrupt handler cancel its registration:

$$\begin{array}{|l}
\textit{IExit} \\
\Delta State \\
\hline
priority > 0 \\
\\
background' = \\
\qquad background \cup \{handler(priority)\} \\
ready' = ready \cup \{handler(priority)\} \\
current' = current \\
handler' = \{priority\} \lhd handler \\
enabled' = enabled \setminus \{priority\} \\
active' = active \setminus \{priority\} \\
\hline
\end{array}$$

Again, the processor must be running an interrupt handler. This handler leaves the interrupt-handling part of the kernel and becomes a background process again, marked as ready to run. As with IWait, the processor returns to the interrupt handler with the next highest priority or to the current background process. The process that called IWait is suspended until the scheduler selects it for execution.

In this schema, the expression

$$\{priority\} \lhd handler$$

denotes a function identical to handler, except that priority has been removed from its domain; it is an injection provided that handler is one.

Two more kernel operations, Mask and Unmask, let interrupt priorities be selectively disabled and enabled. Their specifications are like SetReady, so I omitted them from this article.

The kernel specification is now complete.

Debugging

A useful way to check specifications for consistency is to compute the precondition of each operation and check that it agrees with your intuitions. It was precisely this technique that uncovered an error in the kernel's design.

The IEnter operation as it was implemented was not quite as I specified it. Instead, the kernel designer had tried to combine it with the Select operation that always follows it, like this:

```
┌─IEnterI──────────────────────────
│ ΔState
│ i?: ILEVEL
├─────────────
│ running ∈ background
│
│ background′ = background \ {current}
│ ready′ = ready \ {current}
│ current′ ∈ ready′
│ handler′ = handler ⊕ {i? ↦ current}
│ enabled′ = enabled ∪ {i?}
│ active′ = active
└──────────────────────────────────
```

For efficiency, the kernel's designer combined the two operations, since it made it possible to avoid allocating a special stack area for the kernel itself. But the consequences for the kernel's safety are disastrous. The precondition for the combined operation can be calculated as

$$running \in background \;\wedge\; ready \setminus current \neq \varnothing$$

In other words, the processor must be running a background process and at least one other background process must be ready. If this precondition is not satisfied, the specification says nothing about what happens; in fact, the kernel as implemented goes into an infinite loop, searching for a new process to run, but with all interrupts disabled. Had the implementation kept IEnter and Select separate, interrupt processing could have continued while the scheduler searched for a process to run in the background.

This formal specification documents many important aspects of the real-time kernel. Without delving into details of implementation data structures, it describes not only the events that may happen in the execution of a single process but also how, through these events, a process can interact with other processes and with external interrupts.

But there are several important aspects of the kernel that the specification does not cover:

- It does not model the fact that processes have a state, which must be saved when the process releases the CPU and restored when the process is scheduled again.
- Although this is a real-time kernel, the specification does not require events to happen within any time limit.
- It does not state that processes must be scheduled fairly.

The first deficiency is easily corrected by adding new state variables to model the states of processes and the CPU registers. This makes the specification slightly more complex, but the schema calculus of Z lets the added complexity be separated from the specification as I have presented it.

Fairness is not a property that can be expressed in terms of single events, such as single choices of a process to run, but it must be expressed in terms of sequences — in fact, infinite sequences — of events.

The other two deficiencies are more difficult to correct. A superficial attempt to add timing information might label each operation with the maximum time it is allowed to take, perhaps by incorporating a clock-variable time and adding postconditions of the form

$$time' \leq time + 500\ \mu s$$

to each operation. Unfortunately, this method does not really integrate the timings with the rest of the specification. Also, it cannot express some of the most important timing information about the kernel, for example, the maximum time during which interrupts are disabled. To include such data, you would need to express the specification at a much lower level of abstraction, losing much of the specification's clarity and simplicity.

The third deficiency is that the specification says nothing about fairness. This means that a kernel could satisfy the specification and yet never allocate the processor to a particular process, even though the process was always ready. Such behavior may or may not be acceptable in an application, and in fact many kernels with a priority scheme for background processes are not fair to those with lower priorities.

Fairness is not a property that can be expressed in terms of single events, such as single choices of a process to run, but it must be expressed in terms of sequences — in fact, infinite sequences — of events. The simplest way to specify that the kernel be fair would be to give the Select operation a fixed scheduling order for processes, but that would be tantamount to making the scheduling ring part of the specification, something I was reluctant to do for other reasons.

A more abstract approach is to add an *oracle* to the specification. For example, the kernel state might include an infinite sequence of process identifiers, with the invariant that each identifier occurred infinitely often in the sequence. The Select operation could then discard processes from the oracle sequence until it found one that was ready, and the result would be a fair scheduler that need not have a fixed scheduling order. Implementations of this specification could be justified using the upward refinement technique of He, Hoare, and Sanders.[3]

While all three deficiencies can be corrected, doing so forces changes in the mathematical model behind the specification, making it more complex and difficult to understand. Although the specification as it stands does not express every property required of an acceptable implementation, it nevertheless represents a natural and useful model of what the kernel should do.

This suggests that the idea of a formal specification as a complete contract between implementer and user is not very helpful. A completely watertight contract, here as in the law, is bound to consist mostly of fine print, which can impede the shared understanding on which a healthy

Submit Your Entry for the

1990 Gordon Bell Prize

for Outstanding Achievements in the Application of Parallel Processing to Scientific and Engineering Problems

For more information:
Karen Potes
IEEE Software
(714) 821-8380
soft.one@compmail.com

Deadline: January 2, 1991
Send entries to
10662 Los Vaqueros Cir.
PO Box 3014
Los Alamitos, CA 90720

Prizes of $1,000 each will be awarded in two of three categories:

- **Performance,** based on megaflop rate on a machine with known performance compared against similar applications.
- **Price/Performance,** based on performance divided by the computational engine's cost. Entrants must submit output from a 100x100 Linpack benchmark showing a speedup of at least 5 Mflops.
- **Compiler parallelization,** based on the most speedup obtained through the use of automatic parallelization.

Entries are to consist of a brief (less than 25 pages) report of the application, its implementation, and performance results. Performance must exceed that achieved in previous years to be considered competitive.

IEEE COMPUTER SOCIETY IEEE Software INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC

Computer Scientists

Lawrence Livermore National Laboratory, one of the nation's premier R&D organizations, has a number of opportunities for individuals in our Applications Development Department.

Using state-of-the-art tools and supporting both classified and un-classified scientific and engineering projects, your assignments include real-time software development, image processing, simulation, hardware control/interfacing, software development for celestial navigation, acquiring objects from image data and tracking objects. Other areas include databases, networking computer security, electronic commerce, and combat simulation. Requirements include a BS or MS degree in Computer Science or comparable experience in a related field. Active DOE Q clearance is desired. Must have ability to work well in a team-oriented environment with good verbal and written communication skills. Familiarity with C and UNIX is desirable. Related experience in celestial navigation, target acquisition and tracking, experimental hardware control, mathematical modeling, numerical methods, simulation networking, databases, or Macintosh programming will be considered.

LLNL offers a highly competitive salary and benefits package including three weeks vacation per year. The Laboratory is located in the Livermore Valley, 45 minutes southeast of San Francisco.

Interested applicants should send their resume to: **Patricia Butler, Professional Employment, Lawrence Livermore National Laboratory, P.O. Box 808, Dept. A90320, Livermore, CA 94550.** U.S. Citizenship required. Equal Opportunity Employer.

University of California
Lawrence Livermore National Laboratory

relationship between implementer and user should be based.

If formal specifications have a benefit, it is in raising the level of abstraction, so the answers to important questions — such as whether IEnter and Select may be safely combined — are not obscured by a mass of detail. Questions like this occur naturally in the process of designing the kernel, and reasoning about a formal specification can help to answer them. ❖

References

1. *Specification Case Studies,* I.J. Hayes, ed., Prentice-Hall Int'l, Hemel Hempstead, England, UK, 1987.
2. J.M. Spivey, *The Z Notation: A Reference Manual,* Prentice-Hall Int'l, Hemel Hempstead, England, UK, 1989.
3. He Jifeng, C.A.R. Hoare, and J.W. Sanders, "Data Refinement Refined," *Proc. ESOP86,* (Vol. 213, Lecture Notes in Computer Science series), B. Robinet and R. Wilhelm, eds., Springer-Verlag, Berlin, 1986, pp. 187-196.

J. Michael Spivey is an Atlas fellow at the Rutherford Appleton Laboratory in England and is on leave at the Computer Research Laboratory of Tektronix in Portland, Ore., where he is responsible for developing embedded software for electronic instruments. His research interest is programming.

Spivey received a BA in mathematics from the University of Cambridge and a DPhil in computing science from the University of Oxford. He is a research fellow of Wolfson College, Oxford.

Address questions about this article to the author at Programming Research Group, Oxford University Computing Lab, 11 Keble Rd., Oxford OX1 3QD, England, UK.