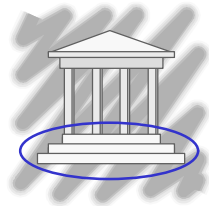


1. The computing platform

The platform is the combination of hardware, firmware, and software that provides the foundation upon which the rest of the M1 can do its job. The platform provides a robust, efficient environment for applications to do their stuff, and abstractions to cushion them from the harsh realities of evolving processor idiosyncrasies.

When our systems go down, there are tremendous potential costs to our customers, and ultimately to Nortel: lost revenue, legal costs, lost customers, replaced equipment, manpower (including sending people to site, investigation, patching, etc.), and potentially even human costs (for instance, when the M1 serves a hospital). While all designers can help minimize software faults, most of the responsibility for containing the effects of faults, providing graceful degradation, or at least recovering from total outages rests on the platform team.

This chapter will focus on the Call Processor platform, for the most part ignoring other software platforms within the M1 system. This is partly because the bulk of the M1 software lives on the CP, partly because the platforms on several of the adjacent processors might reveal similar lessons, and partly because I don't know anything much about the other ones (like MAT, SCCS, EIMC, etc.).

1.1 What's in a platform?

The platform doesn't really contain anything customers know they want. Its job is to enable designers to create applications which live up to the pervasive requirements discussed in Chapter 2. A few ingredients are critical:

- **Multitasking** – (but with fast context switches) to simplify designs
- **Mutual exclusion** – to prevent other tasks from running some of the time
- Suitable **scheduling** – to help the tasks work sensibly together
- **Redundancy** – so the system is always available

- **Fault handling** – some degree of fault tolerance, plus diagnosis and (self) repair
- Cold/warm **restart** – for initial installs, and when fault handling doesn't suffice
- **Communication** with other local nodes – synchronous and asynchronous
- **Timers** – lower overhead delays and wake-ups, plus reliable time of day
- **Memory management** – including hardware protection
- **Device drivers** – disk, tape, terminal, and maybe others
- System **configuration** – because we support many different setups
- Live **software upgrade** – so that we can install new features without causing outages
- **Patching** – so we can fix the new features
- **Debuggers** – hopefully including a set of field-safe tools
- **Reliability** – because our customers have no tolerance for down-time
- **Low overhead** – both in terms of real-time and code bulk
- **Bearable cost** – because customers want to buy features, not platforms

Some other services are just "nice-to-haves", because we can work around their absence or create them ourselves on top of the platform:

- **Third-party software** – the larger the available selection, the better
- **CORBA** – to let the third-party stuff talk to each other
- **Quality of service** selectability for control:
  - Reliable, Flow Controlled (Slow)
  - Try-once, Overload Controlled (Fast)
- **Name mapping**
- Protocol versioning

And a few services which are standard on some operating systems we would prefer to live without:

- **GUI libraries** – our GUIs are all off-board for performance reasons
- **Garbage Collection** – none are quite fast enough (yet)
- **Web Browsers** – gimme a break

Not surprisingly, we have cobbled together a system that delivers reasonably well on the must-haves, if less convincingly on the would-be-nices, and avoids the rather-nots. Keeping the cost bearable and the code bulk down will probably always mean not having all of the bells and whistles.

Although our platform is now based on VxWorks running on an MC680x0, this should be viewed as the *current* commercial platform, rather than the only or final solution. As designers, we should expect to see at least two, and possibly three platforms in the near future. Nonetheless, real-time code, even good, OOed, layered, concerns-all-separated, real-time code, is ultimately not vague enough to let us completely ignore the specifics of our platform. The following sections attempt to impart a broad understanding of how VxWorks' idiosyncrasies affect our designs, and also how our code might be written to also allow reasonably painless porting to other OSs.

Why should we even have a platform team?

Years ago, we built a processor which was so darned unimprovable we called it "Omega". Right. That was four generations ago, and it seems unlikely we will ever get a "last step" in processor evolution, but it *was* the last time we tried to have a CPU built just for M1. Somebody noticed that there are other organizations (eg: Intel, Motorola, etc.) whose CPU production volumes and corresponding R&D efforts are so much huger than ours that it seemed silly and unnecessary to compete with them.

The next logical step was the operating system: if we don't run on a scratch-built chip, why do we need to have a home-brewed Real-Time Operating System? The answer turned out to be that lots of commercial operating systems were a poor fit for our product. We need our platform to be small, fast, and fault-tolerant. It is perhaps good news that hardly anyone fills our needs perfectly, but we *are* now able to buy pieces of the solution that do a lot of the work for us. The theory goes that we can then tune them, or maybe layer things on top of them, to end up with a good enough platform that has two strategic advantages over the old one: it's cheaper, and we can buy some applications off the shelf which will already run on them.

By induction, we could simply buy each successive layer, including the applications, and all go surfing. The good news (job-security wise) is that we seem to be good enough at solving some of the market requirements that it's worth keeping the R&D shop open. However, more and more of our work will leverage externally-designed products.

1.2 Vanilla VxWorks

Wind River Systems provides pretty good VxWorks documentation. In particular, the *Reference Guide* gives an exhaustive API description, and the *Programmers Guide* gives instructions on how to use it. The Training Workshop notes are also good, but are best understood with the accompanying lectures. Unfortunately, collectively these three documents are about six inches thick.

This section is an attempt to get you started with a lower overhead. It's not a substitute if you need to be an expert, but it's probably good enough for a typical designer. For the next level of detail, see *Inside Thor*.

This section attempts to distill the most crucial VxWorks[1] essence into a manageable volume, and adds some specific rules for use within the M1 context and a few juicy stories about how things have gone wrong when these rules weren't followed.

VxWorks is a commercial real-time operating system suite which runs in (among other things) JPL's *Pathfinder* Mars landing vehicle, and the Meridian 1. Not unlike the nuclear missiles in which it also runs, VxWorks was built for speed but not comfort—latency is known and controllable, but major design decisions consistently favored performance over safety.

VxWorks is based on preemptive priority multitasking, simplifying control of concurrent transactions by allowing solutions to mirror the real-world problems they're modeling.

The VxWorks kernel is a set of normal subroutines, as opposed to the other common types of kernel: supervisor mode subroutines, a set of tasks, or a fried chicken pitch-man (oops, sorry). In VxWorks, all tasks run in supervisor mode (remember, performance over safety). Therefore, application code and Interrupt Service Routines (ISRs) can invoke the kernel with normal subroutine calls, keeping the overhead extremely low.

Even though we now (since CP2/Rls21) use VxVMI to manage memory protection, tasks still share a single flat address space. This gives us fast inter-task messaging and fast context switches, but it means bad pointers in one application can end up trampling store belonging to any task. Also, there is no explicit detection of memory leaks, and certainly no garbage collection (speed, not comfort).

1.2.1 Tasks

Task implementation is also very light-weight: each task is represented by a Task Control Block (TCB) and a stack. Apart from the full register save, task context switching is not much more expensive than a subroutine call. On the other hand, stack crashes are not detected (yet again, performance over safety).

[1] For the curious, the name "VxWorks" comes from the early days when Wind River made a toolkit that "worked" with Microtec's VRTX real-time operating system, although VxWorks no longer contains any of the original VRTX code. Nobody admits to knowing what VRTX stands for anymore…

Why might we prefer stack overflow detection?

Outside of North America, public dial plans are seldom the regimented 3+3+4 digits we're used to seeing, and variable-length directory numbers are common. A site in England (let's call it Big Important American Bank) did a configuration change to handle International digit processing. The problem was that the software which does translations is recursive, and the depth of recursion is dependent on the numbering scheme, so that even though it tested out fine at home, it blew up in the field.

It gets worse. The stack overflowed into protected memory, and the processor simply stopped dead during the next recursion when it tried to write to the stack. The watchdog then started barking, and the CPU did a forced switchover. This happened erratically, but often, so it looked like we suddenly had huge hardware problems. Tracking this down to its root cause ended up being very expensive.

Moral: If you create a new task, ***make sure to allocate a generous stack***. Not only will it be used to store all of the locals and registers for your own procedures, but also for any operating system procedures that you invoke. Worse than that, most interrupt handlers also use *your* stack space. So allocate what you think is more than you need, test your code thoroughly (ideally under a reasonably heavy traffic load) and then use the VxWorks `checkStack` tool to make sure there was still a comfortable buffer of unused space at the top of your stack. Remember, stack overflows aren't detected, so if you get this wrong, *your* code will still work. You'll just have trampled somebody else's data, which gets very tough to debug. A bigger stack doesn't cost any more real-time, and memory is cheap.

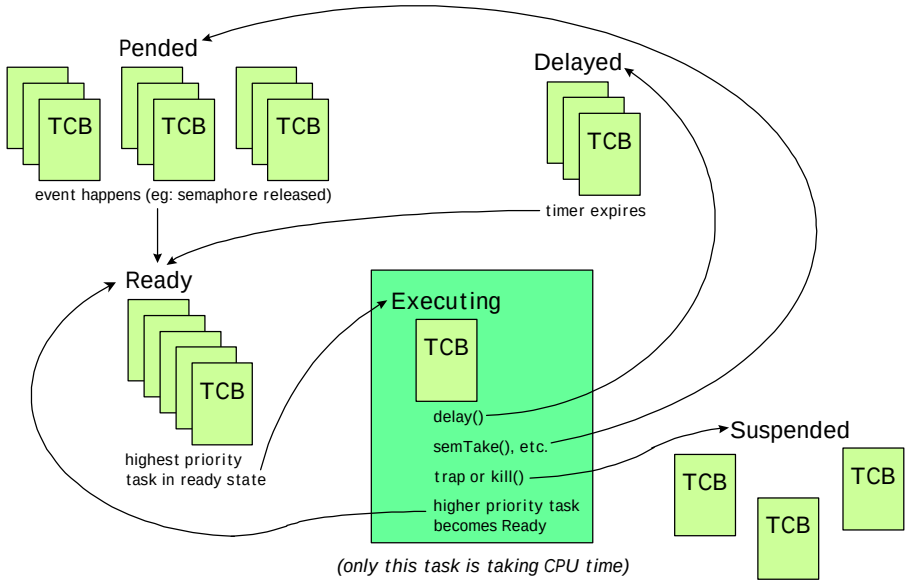
Bonus advice about stacks: Never return a reference to a procedure's local variables. Locals are kept on the stack, which gets reused by other procedures after you return. The simplest case actually works: in the absence of interrupts, the procedure which just called you can see anything you left on the stack, as long as it hasn't called any other procedures yet. This is sort of a curse, because it leads to transient errors, rather than simply failing utterly every time you test it. Returning pointers to local variables is a good problem to look for in code inspections.

As a rule, the best time to create tasks, or any other system resources, is on a system restart. On any real-time critical system that may run for years without taking a break, it's better to have a reusable pool of each resource type that's a bit bigger than you need, than to keep allocating and freeing resources. The same applies to memory allocation. (See the Memory Manager pattern in Chapter 3.) M1 software mostly follows this guideline, but the problem is that we use some third-party software that doesn't. In particular, the TCP/IP stack and the ORB both do dynamic `mallocs` and task creation, and have therefore caused some performance problems and memory leaks.

The old SL-1 code now runs as a single self-contained VxWorks task, `tSL1`. Within this task, `WORKSHED` schedules the SL-1 transactions as if they were being handled by a number of separate tasks. See the next chapter for details.

1.2.1.1 Task state transitions

VxWorks allocates CPU time to tasks in a very predictable manner. Except during interrupts, the highest priority task that's been in the ready state longest is always the one running. When application code or an Interrupt Service Routine (ISR) does anything that might make a higher priority task ready to run (such as giving a semaphore) the corresponding kernel subroutine ensures that the right task will be run. ***All*** state transitions happen as a side effect of kernel subroutine calls by tasks or interrupt service routines. If two or more tasks have the same priority level, the tick ISR coordinates round-robin sharing among them.



When a task traps, it moves to "suspended" state. VxWorks provides a "delete hook" option, which lets you invoke a procedure to do any necessary cleanup whenever a task traps. Resources like file descriptors, semaphores, sockets, and memory blocks are freely accessible among tasks. This flexibility means that our coding policy must discourage indiscriminate sharing of resources. Since VxWorks doesn't, task code ought to keep track of any resources it owns and free them using a delete hook. At the moment we actually restart the system for `tSL1`, which is an effective if slightly heavy-handed way of ensuring that all operating system resources are cleaned up.

You can end up giving a semaphore to a suspended task if you're careless and VxWorks won't notice (but the watchdog probably will…).

1.2.1.2 Task priorities

Priorities are assigned on a stable basis (with the new exception of `tSL1`). The conventional wisdom is that this is fairly easy to tune and more efficient than trying to compute algorithms like "run the task whose deadline is earliest".

"Priority inversion" is when you have a high priority task waiting for a lower priority task to complete its work. It can happen during semaphored sections, but VxWorks provides a parameter, `SEM_INVERSION_SAFE`, in its `semTake` procedure which will prevent inversion. The trick is to temporarily set the priority of the running task equal to the highest priority task which is waiting. See the *VxWorks Programmer's Guide* for a clear example. We end up sort of doing this manually for the SL-1 task because it doesn't schedule its work queue using VxWorks semaphores. Also be aware that message queues can suffer from similar problems, which need to be planned around.

Task priorities are determined by black magic, but there are some heuristics to help. In general, more important and shorter deadline stuff needs to be high priority (closer to 0). It's also generally better to have a server process running at a higher priority than its clients. Even though VxWorks lets you, it's usually best not to change running task priority because it gets very hard to maintain deterministic behavior.

> **Struggles with dynamic priorities**
>
> In Rls18, we wanted to get fair-share scheduling. We built it, but it interfered horribly with the VxGDB debugger. Early versions went through and removed all breakpoints on *every* task-switch interrupt, had a huge overhead, and sometimes artificially elevated the priority of a pended or even suspended task. So although it was almost working, we ripped it out before shipping. In Rls22, we redid it, but now at message interrupt time, not on clock ticks. The goal is to have a transaction engine with priorities appropriate to the current transaction. The Interrupt handler puts a priority on the message in the queue, and the ISR changes the `tSL1` priority to do proper priority inversion. After another recent effort, we now change the priority of the `tSL1` task dynamically, again to try to match transaction priority with task priority. And yet another effort went into Meridian Evolution. We'll get there in the end…

Within each priority, we have enabled round-robin CPU sharing by calling `kernelTimeSlice`. The problem is that then round-robin scheduling happens within *every* priority (that is, all tasks at priority *x* will share time equally amongst themselves, all tasks at priority *y* will also share time equally amongst themselves, etc.). A more controlled but time-consuming way to accomplish something similar is usually to call `taskDelay(0)` to swap explicitly with any other waiting task within your priority. This would also allow you to determine when the swap

happens, and avoid things like `tUsrRoot` sharing with `tExcTask` in a non-deterministic manner. On the positive side, our method is a cheap way to almost get class-based scheduling. On the negative side, you have to make sure you play nicely with other tasks at your priority level—don't trample each other's data.

The following is a catalog of the tasks running on a Meridian 1 switch, their priorities, and a rough description of what each is for. Of course this list changes over time, but it should give a sense of what's going on. The "`i`" command from the VxWorks shell will list the tasks running on any given load you're interested in.

Task Name	Priority	Purpose
[tUsrRoot]	*[0*]*	*[the initial task: configures the system, spawns the shell, then exits]*
tExcTask	0*	exception handling
tLogTask	0*	message logging and output
tSwd	0	software watchdog
hiExcTask	1	additional exception handling for HI
tRstTask	11	restart logic: registers task starts, restarts, deletes
tRpt	11	writes the report log to disk
tTimer	11	MAT: SNMP heartbeat task
tEvtColl	11	MAT: alarm management event collector
tEvt	11	"
tRdbTask	20*	VxGDB host debugger task
tMMIH	21	MSDL/MISP interface handler
TimerThread	40	Mobility: ORB timer heartbeat task
tNetTask	50*	task-level network functions
tOrbixd	50	Mobility: Iona's ORB
tFtpdTask	55*	FTP server
tTftpdTask	55	TFTP server
hiserv0	60	HI utility tasks, handles work q'd by HI ISRs
hiserv1	60	"
hiserv2	60	"
hiserv3	60	"
hiExcScan	60	scans for exceptions on HI-managed devices
cnipMon	60	monitors CNI ports
ipbMoni	60	monitors cards on the CPU backplane

tTapeTask	60	tape emulation for legacy database interface
RootTmrMgr	80	Mobility: timer server task
tRlogInetd	100	rlogin daemon
tPortmapd	100*	RPC port mapper
pdtLogin	100	direct (serial) PDT login monitor
pdtBrkTask	100	PDT breakpoint handler
tDTPreaper	100	old Data Transfer Protocol (should be removed!)
tDTPlisten	100	"
BootpServer	150	Mobility: TCP/IP bootp server for EIMC
hifmon	230	HI hardware fault monitor
tod24	240	24-hour time-of-day, manages non-SL1 "midnight" audit jobs
tSNMP	240	MAT: Simple Network Management Protocol
tScriptMgr	240	MAT: maintenance windows
bootpSrvrd	240	Mobility: alternate bootp server (orb-based?)
mspServer	240	Mobility: Mobile Service Provider (call proc.)
tPRNT	240	MSDL/MISP card interface handler
thlpTask	240	overlay supervisor-accessible help feature
pdtShell01	240	direct-login PDT shell
tRlogind00	240	MAT: network login daemon
pdtShell02	240	MAT: network login PDT shell
tRlogchl00	241	MAT: network login PDT child
tAlarmLog	245	Mobility: alarm management
OAMSRV	245	Mobility: OA&M server
tSL1	250**	The SL-1 code, basically all of call processing

* VxWorks predefines these priorities

** the SL-1 task changes its priority depending on what it's doing

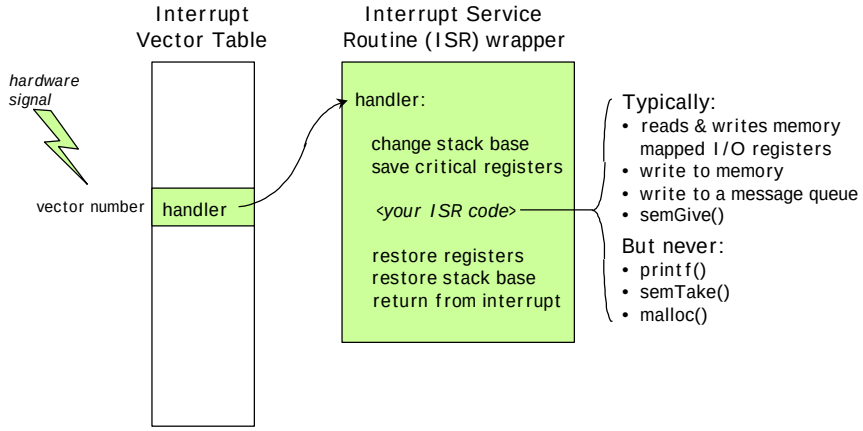
> **Why is `tSL1` the *lowest* priority task?**
>
> If the most important job of the PBX is call processing, you might expect `tSL1` to be much nearer to the top of the list. The problem is with `WORKSHED`, the SL-1 work scheduler that acts as a mini-OS task manager. It controls a number of independent activities, some of which are audits that run whenever there is nothing better to do. These will take an infinite amount of time if they are allowed to do so, so nothing with a lower priority than `tSL1` can ever expect to get CPU time. To make call processing work in spite of this, we sometimes try to change the `tSL1` priority dynamically, but the safest technique is to ensure that no other tasks take very much of the total CPU time, and that what they take is split up into very short segments.

1.2.2 Interrupts

Interrupts are requests for urgent, *short* pieces of work to be done. They signal the CPU to halt normal task processing (whatever the priority) and call the routine pointed to by the interrupt vector table. They are predominantly used to service hardware (clocks, device drivers, DMA) and handle exceptions (like the bus errors caused by referencing illegal addresses).

If you have occasion to write an Interrupt Service Routine (ISR), there are a number of special requirements to consider.

- Write very short, very fast, very clean code. Code inspect it thoroughly. It's extremely important that all interrupt code be very reliable. A bus error will cause an ungraceful switchover or a warm restart.
- Don't process a message in the ISR. Just enqueue it against task code, and return. Do the processing as part of standard, prioritized task execution.
- Remember that there is no "context". When you get called, the registers contain things that are important to whatever code you've just interrupted. If you're doing anything at all complex (and you probably shouldn't be!) you'll need a wrapper around your code:



- Finally, remember that the MC68k series chips use the task stack for processing interrupts, so people have to allocate enough space to handle the worst-case nesting of interrupts on top of normal task recursion. If you need a lot of stack space (and you probably shouldn't) you should reset the stack base to a safe area of memory that you have previously allocated, and then restore it before returning.

1.2.3 Types of memory

At loadbuild time, VxWorks allocates three types of memory: text, data, and BSS. "Text", per Unix jargon, is where your executable code is stored. Data segments contain text strings and other large constants. Blocks Started by Symbol (BSS), are things like arrays and static C variables. Text and Data comprise the load file. BSS is not loaded, but is zeroed out during the boot process.

The rest of the memory we use is allocated dynamically (via `malloc`), usually during a restart. There is no simple way to allocate memory for use by a task in such a way that it would automatically be deleted if the task goes away (although if the task traps, and we go into a restart, all the unprotected memory is cleared anyway ☺). Memory protection and restarts are covered in more detail in Section 5.3.1.

> **WARNING:** When different VxWorks tasks call the same procedure, they get *the same* text, data, and BSS segments (unlike in Unix, where only the text segment is shared). Their stacks will of course be different, so code reentrancy is possible but it's not automatic. (Speed is better than safety.)

1.2.4 Posix

Portable Operating System Interface for Unix (Posix) is the IEEE's attempt to be a universal, Unix-like Operating System API. VxWorks supports a Posix API (along with many non-Posix extensions). If we used it exclusively, it would make porting to some other operating systems almost effortless. However, it appears at present as if some of the proprietary parts of the VxWorks API are too helpful to ignore. But if you have the choice, stick with Posix.

1.2.5 Assorted VxTools

The following is a list of more-or-less useful tools available on VxWorks:

VxHelp	gives detailed on-line help for the following stuff
the shell	can interpret C code to query values of variables, etc., can also set breakpoints (but usually use VxGDB)
`moduleShow`	find out what VxWorks knows about a loaded module
`lkup`	look up symbols containing a given substring
`i,ti`	display information about one or all tasks
`tt`	trace a task's stack
`s,so`	single step
`b,bd,bdall`	set and remove breakpoints
`l`	disassemble code
`c,cret`	resume execution of a task
`sp,td`	spawn or delete a task
`ld,unld,reld`	load, unload, or reload a module, not usually used by us (try patching)
`timex,timexN`	times execution of a function
`spy`	gives a task activity profile to see who's hogging the CPU
WindView, Stethoscope	Look useful, but not currently used much by us. Maybe a good job for a coop?

1.2.5.1 GNU source-level debug (GDB)

This tool is allows debugging of the C/C++ portions of the M1 software to be debugged at the source-code level. It has a more limited ability to understand SL-1 code, and is not available in the field.

See T00057, *THOR High Level Debugger* in Doctool library TOOLDOCS, or try http://47.82.33.147/~mtvjbg01/SDE/GDB_MANUALS/Content.html

1.2.6 Compiled caveats

The following catalog of hints and warnings should prove useful to designers doing any detailed interaction with the operating system:

- The shell does not understand symbolic macros (`#define`) so use grep to find the corresponding value.
- The shell interprets all variables as 32-bit integers unless otherwise specified.
- The shell doesn't really understand data structures (use VxGDB).
- TaskIds are unique at any given point in time, ***but*** they get re-used when the task dies!
- VxWorks does not check to see if a directory is valid on a shell *cd*
- Task variables cause context switches to be slower
- If you use `semFlush` to "catch up" all tasks waiting for a semaphore, the semaphore state still does not go to full.
- Mutual exclusion semaphore's may not be used in an Interrupt Service Routine. Use `intLock` instead, but use it cautiously. `intLock` does not prevent task context switching if you either block (eg: `semTake`, `malloc`, etc.) or unblock a higher priority task (eg: `semGive`), and interrupts are unlocked on every context switch.
- If you're using a mutual exclusion semaphore (or writing an ISR for that matter), don't do any work which could be placed outside of the critical region, don't `pend` or `delay`, probably don't even loop… This will help avoid latency problems, deadlocks, and errors. To help this be true, you probably want to put an API on the routines which access your critical

resource and initialize and manipulate the semaphores yourself rather than having your client code manipulate them directly.

- Fast event occurrences can cause lost information if the task waiting on a semaphore is not high enough priority. It is instructive to think through the following example. If VxWorks is executing this code:

```
FOREVER
    {
        semTake (semId, WAIT_FOREVER);
        printf("Got the semaphore\n");
    }
```

then you will have the following behavior:

```
> semGive(Semid)                                  → 1 message printed
> semGive(Semid);semGive(Semid)                   → 2 messages printed
> semGive(Semid);semGive(Semid);semGive(Semid)    → 2 messages printed
```

If you increased the priority of the server loop, you would get the expected 3 messages printed, or you could use a counting semaphore.

- When using `msgQreceive`, if the message in the queue is 75 bytes long but you only ask for 50, the remaining 25 bytes are lost permanently.

- Message queues end up copying the data in your message at least twice, so keep messages smallish. For better performance (especially from an ISR) consider putting pointers to the data into the message, rather than the data itself.

- If you're worried that you may not own all accesses to a critical resource, you can use `taskLock` to protect your critical region, but again, keep it short.

- To avoid deadlocks, try using only a single semaphore to protect resources which will need to be accessed at the same time. Failing that, apply semaphores in a strictly hierarchical, nested manner. That is, have a master lock that controls access to the lower-level locks. Grab the master, try to acquire the lower locks, and if one is unavailable, free all the locks you've gotten so far. (This is of course a lock management transaction engine, and is good general advice, rather than a VxWorks-specific trick…)

- `malloc` can be slow, and may even pend under certain circumstances.

- `taskDelay` is subject to drift. For more accurate task start intervals, try `wdStart` with `semGive`, because this way your next timer is restarted during

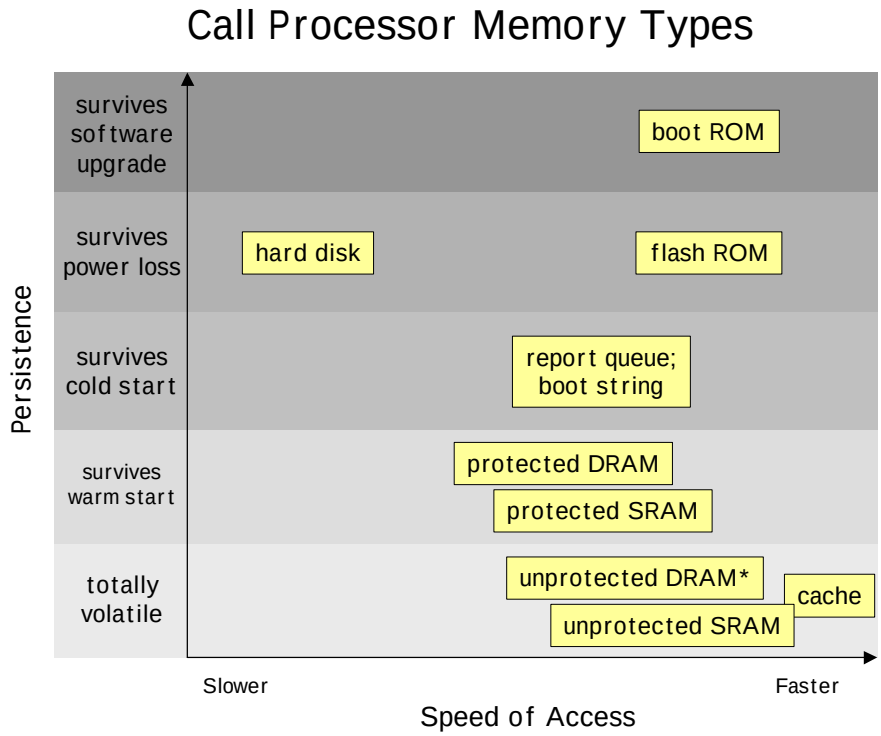
the ISR, rather than when your task actually gets to the front of the ready queue.

- The Standard I/O Library contains macros as well as routines. As usual, breakpoints can't be set in macros.
- Mobility, SMP, ISDN, QSIG, MMIH, and ICCM all now use the OS heap.

1.3 High availability: Our modifications to VxWorks

1.3.1 Robust memory

One of our first extensions to the VxWorks platform allowed us to engineer what sort of memory gets used to handle different things. The following details change with every generation of hardware, but the considerations remain valid. Most of the details are carefully hidden from application code by the platform team.



The **boot ROM** amounts to a very short program that only knows how to find and load the main software. It is shipped on the CPU board, and is never expected to change. It must be short enough that it is provably correct; it more or less can't be fixed.

Flash ROM is relatively cheap, very stable, and fairly fast to read. Write access is slow and awkward. It turns out to be a good place to put the executable code, but it does make patching a hassle. It also makes it hard to set breakpoints (although in the lab we don't usually put the code into flash). We use the MMU to do set patches and breakpoints, temporarily mapping what should be a flash address to a spot in RAM. Because you always want a complete, valid copy of the OS *somewhere*, flash is duplicated, and we only update one side at a time.

Dynamic Random Access Memory (**DRAM**) is cheap, and access is fast. On redundant machines, there is a complete DRAM bank associated with each CPU. The active CPU updates its DRAM, and the Changeover Memory Bus (CMB)

ASIC copies these updates to the other DRAM. Besides the cost of the hardware, there is also a 2× performance cost on writes to shadowed DRAM.

The report queue is required after a restart so that you can reconstruct what went wrong. The boot string is a series of parameters that help the boot ROM determine what the best place to find the boot image is likely to be. Both are stored in a magical area of DRAM that is not trampled during a restart or even a reboot.

Static RAM (**SRAM**) is not as cheap, but is faster than DRAM. We often provide at least a small amount of SRAM to try to improve system performance. With the right tools, it is likely that we could get better performance from our existing hardware by carefully tuning what we put into SRAM.

Finally, there is usually on-chip **cache**, and sometimes nearby L2 cache, for both executable code and data. Cache is flushed at run time on a simple least-recently-used basis. Because our code tends to branch around an awful lot, we typically choose very short line sizes for the cache. Even so, neither data nor program store caching is as effective for us as it is for some types of problems. The current cache is write-through, and takes roughly 7 times as long to write as to read.

A **hard disk** or other commercial mass storage system is sometimes used to store things like the customer database. Disks are very cheap memory, but access time is non-deterministic, because it depends on things like where the disk heads are, so we can't use them to store data we need in real time.

A further level of reliability, already discussed in Chapter 2, is provided by using the MMU to make certain areas of RAM **protected** against inadvertent corruption. We require people to explicitly unprotect data, change their variable, and then reprotect it. Of course, during this window, they are also free to trample anybody else's protected memory, so it is very important that they put some care and attention into this part of their code. The unprotect/reprotect sequence costs a bit of real-time on writes, but there is no extra cost for reads. As a rule of thumb, state data (like Call Registers) that is subject to constant churn is in unprotected memory, while configuration data (like the customer database) is in protected memory. More surprisingly, we also store analog trunk states, which change constantly, in protected memory. This apparent eccentricity is to ensure that we don't end up hanging these trunks over a restart.

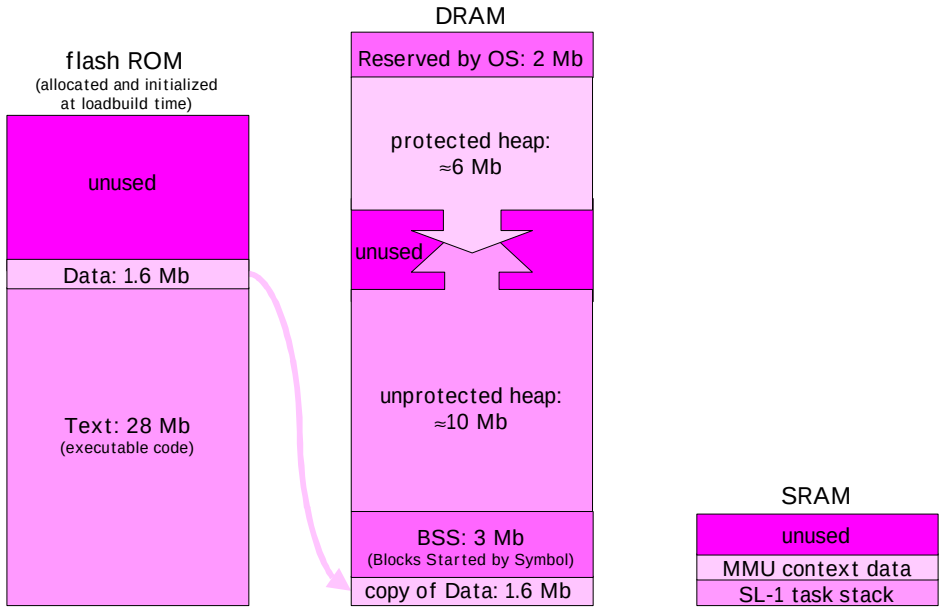
Protected memory isn't handled very easily for third-party software—if it needs to update protected memory, we have to turn off memory protection globally, invoke the third-party software (which may be interrupted by higher priority

tasks), and turn protection back on when it returns, which is obviously a bit unsafe.[2]

[2] This caused a major performance problem for Meridian Evolution, when the hardware changes forced a change in our memory protection software. Essentially, where `UNPROT` had once meant "toggle a bit on the processor board", it now meant "scan through the MMU table and toggle the setting for all pages that are currently protected". This in turn slowed restarts, which do a lot of protected memory updates, down to a crawl. It's an example of the dangers of using the wrong algorithm in a low-level routine. The eventual solution was to use a pair of memory protection maps: one with some pages protected, and one with everything unprotected. Then when `UNPROT` is called, simply swap out the real table.

1.3.1.1 CP memory layout

The detailed layout of memory on our machines is very dependent on node type, number of telephones, hardware generation, type of SIMMs installed, and software version. The following picture, based on the first Meridian Evolution release, is presented only to convey a general sense of what's out there.



The most recent detailed memory layouts are available from the performance group in Mission Park. At the time of printing, the best document was probably Marjie Hempstead's *Meridian 1 System Capacities, X11 Release 23*, in Doctool library SL1DOCS, although *Inside Thor* also has some good details up to CP2.

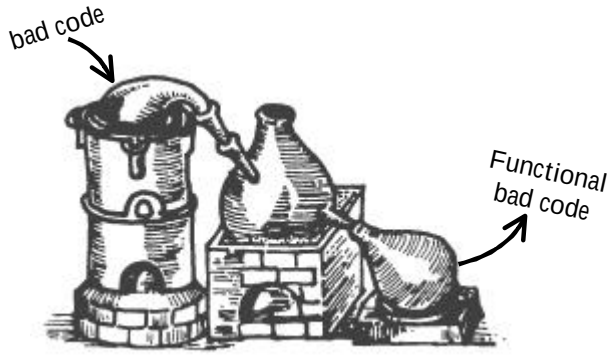
1.3.1.2 Conjuring with bad pointers

If somebody tries to write to Flash ROM, we ignore it. The theory is that our only easy alternative would be to trap, and do a warm start. What has probably just happened is that somebody has dereferenced an uninitialized local pointer. Locals are stored on the stack, and one of the most common other things to find on the stack is a return address, which is just a pointer to program store, which is in Flash.

Now of course the phone call in question will probably not behave the way the designer would have wished. If the purported pointer was supposed to refer to any very interesting data, the odds are that the subscriber will end up having to hang up and redial, but at least we didn't cause the switch to restart.

A related trick is called "pointer remapping". If we attempt to write to a totally invalid address, we catch the exception, map that address to a special area of RAM, and store the value there. Subsequent writes or reads at the same bad address will also be caught, and we will return the value we put there. If the first bad reference is a read, we set up the mapping and return zero. The alchemy sort of works. We take bad code and may make it function correctly. Of course, it's still bad code. We do generate a record of the problem and encourage people to debug their pointer setup, but it does provide complete symptomatic relief some of the time.

While both of these tricks prevent outages in the field, they also mean that fewer bugs get noticed (and fixed). It's a philosophy predicated on living with bad code, rather than working towards perfect code. We might be better off to hide the errors only in the field, so that at least in the lab we get the failures that force us to investigate the errant code.



pointer alchemy apparatus

We may soon stop doing pointer remapping. Instead, a bad SL-1 pointer would signal a longjump back to the start of the WORKSHED loop. Other tasks would be killed and restarted after suitable information has been captured (see *Requirements Specification for Exception Processing* on the Platform Team's home page— http://47.82.33.147/projects/OSEvolutionSite/).

1.3.1.3 Virtual memory

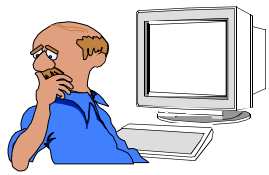
> **Wind River on Virtual Memory**
>
> The VxWorks belief system holds that:
>
> 1) all threads shall run in a single flat virtual address space, and
>
> 2) the OS shall be "just a collection of libraries" that an application links to as if they were any other libraries.
>
> By having a flat address space, addressing is faster generally, and in particular caching may be more efficient across context switches by allowing you not to flush the cache. As with most embedded systems, VxWorks provides no true virtual memory. That is, you can't swap pages to disk to fake a bigger more usable memory than the RAM you actually have. Virtual memory is nice for desktop systems, especially when the cost of 1M of RAM is much greater than the cost of 1M of disk swap space, but it leads to slower, non-deterministic performance. So the conventional wisdom is that you would never want VM for an embedded system (although this is exactly what the original SL-1 did with "overlays", because they are the non-real-time side of our real-time system).

We do sometimes use a form of virtual addressing to implement our patching strategy. Because the code is running in flash ROM, which we can only change in 256K chunks, we don't do in-line patching. Instead, we put the patches into regular data store, and then tell the MMU to make sure we execute the patched version of the procedures.

At several times during the history of development, the amount of available memory has been extremely tight, or CP performance has been bounded by memory throughput. Both problems have led to people going to a lot of trouble to optimize memory usage, and this shows up in some densely-packed, multiply-overlaid structures, such as Call Registers.

1.3.2 Robust mass storage

Commercial disks are cheap and already fairly reliable, but we also sell redundant disk configurations (sort of minimalist RAID systems) to ensure that customers' databases are preserved. Of course, this is no protection whatsoever against either bad software corrupting the image before it gets saved, or operator



error. (“Dang! I just deleted the master archive file again…”) The Mass Storage Redundancy (MSR) feature allows duplication of either directories or files, so that methodical organization can prevent such catastrophes.

The M1 hard drive is partitioned into three directories: /p, /u, and /id0. These contain protected files (software, firmware, default database files, report data text file, script file, non-customized files); unprotected files (database files, error reporting files, patches, files generated by the system during run-time, customized files); and card id files, respectively.

The report log tracks the operation of the system, and records any abnormal conditions. A typical file would contain records like the following:

```
500 : SRPT0770 TOD 1: Midnight job server starts on side 1
       Number of jobs to do: 2 (15/9/93 2:00:00.975)
501 : SRPT0773 TOD 1: Starting midnight job ‘rstThr’ (15/9/93 2:00:00.979)
502 : SRPT0773 TOD 1: Starting midnight job ‘pchMidNite’ (15/9/93 2:00:00.981)
503 : SRPT0774 TOD 1: Midnight jobs completed on side 1 (15/9/93 2:00:50.487)
504 : CIOD0157 CMDU 1 is ACTIVE, RDUN is ENABLED (15/9/93 2:12:02.516)
505 : HWI0009 HI FS: saving data to directory “/u/db/hi_bak” (15/9/93 2:13:01.133)
506 : CCED0760 SWO 1: Graceful switch-over to side 0 requested (15/9/93 3:14:46.615)
507 : HWI0003 HI Init: Graceful SWO Start continues on side 0 (15/9/93 3:14:24.647)
508 : HWI0004 HI Init: Phase 5(“objects link”) begins (15/9/93 3:14:24.647)
509 : HWI0004 HI Init: Phase 7(“objects enable”) begins (15/9/93 3:14:24.092)
510 : HWI0007 HI Init: SWO  Start complete at side 0 in 0 seconds (15/9/93
       3:14:25.674)
511 : CCED0762 SWO 0: Graceful switch-over to side 0 completed
       Previous Graceful SWO: at 14/9/93 3:15:03 (15/9/93 3:14:25.861)
512 : BERR0705 EXC 1: Bus Error in Task “tSL1” (0x4710000)
       SR=0x3000, PC=0x46d758a, Addr=0x1670fc40, SSW=0x074d (14/10/93 14:32:27.663)
etc.
```

Unlike some call processing systems, we do not use our disks to store Call Detail Records as they are generated. Instead, they are buffered (using CR data blocks) until they get shipped to an off-board processor, usually over a narrowish-band dialup port. We have had systems run out of buffers because the port speed (eg: 1200 baud) was too slow to keep up with their call traffic, and this caused the extra CDRs to be tossed. There is an opportunity to improve on this, especially in the low end of the market. On large systems, it’s probably just as well that we process off board.

For a more detailed description of the Core Multiple Drive Unit (CMDU) layout, see *Inside Thor*.

1.3.3 Watchdogs

As discussed in Chapter 2, a watchdog is a timer designed to detect deadlocks, infinite loops, and other situations where the switch “hangs”. Each board has a hardware watchdog timer on it. The watchdog hardware simply counts down to

zero, and as soon as it gets there it signals a switchover (on redundant nodes) or a cold restart. What keeps this from happening is that the software watchdog task, which runs at priority 0, and loops continuously like this:

```
DO FOREVER
     check_that_all_tasks_are_healthy;
     2_second_delay → hardware_watchdog_timer;
     DELAY(less_than_2_seconds);
```

Because the software watchdog runs at the highest task priority, we know that if the hardware watchdog time expires ever, no software is running.

The software watchdog also keeps track of how long other tasks are running, and may trigger a warm restart if it thinks there is a problem.

1.3.4 Restarts

When things go badly wrong (a task traps, too many pointers are remapped, a watchdog timer expires) we can't just halt and display "`An unexpected error has occurred`" on all the phones. Most people don't even think of a PBX as a computer, and they have no patience for the sort of bugs they have learned to expect from computers. As mentioned in Chapter 2, what we usually try to do is restart the system in some way, but causing the minimum amount of disruption needed to fix any particular fault. There is a progression of restart types available, and if the first type doesn't fix the problem, we'll usually move on to the next.

1.3.4.1 Task restarts

Individual VxWorks tasks can be created and killed independently. As with all resources, we prefer to do this only at the time of a system restart, but there are at least two other times it happens. The first is that if a new user logs in, a task is created to serve the terminal, and this task is deleted when the user logs out. The other major case is when a task dies violently. Under normal circumstances, we would then recreate the task immediately.

Until recently, this applied to the SL-1 task. That is, if `tSL1` choked or if the software watchdog expired, we would recreate the task, call `WORKSHED` which in turn would call `INITIALIZE` to set up all the dynamic call processing data structures, and carry on. This was known as an "init", and used to be the fastest of the various M1 restart types. `tSL1` task restarts have recently been disabled because we had trouble keeping the state data synchronized between the newly-initialized SL-1 code and all of the other tasks, but they may return again in the

future. For now, we just go straight to a warm restart, although the other tasks can still be restarted individually.

1.3.4.2 Warm restart

Warm restarts happen if the manual reset button on the CP card is pressed, an interrupt handler traps, or we get too many pointer remaps. We restart the operating system, which means that all of the unprotected data gets deallocated, including the Call Registers. We then run the initialization code in SL-1 module INIT, which sets up all of the basic SL-1 data structures and then queries the network connection data to set up the appropriate CRs. This process does not manage to recreate all of the subtleties of call feature data, but basic POTS behavior is preserved. Calls which were not in a talking state (and thus had no network connections) are dropped. No new call processing transactions are processed until this phase has been completed.

Warm restarts also now rebuild the ACD queues, using some unprotected data that is salvaged before the operating system gets a chance to clear it.

> WARNING: Protected data survives a warm restart, but unprotected doesn't. It is critical that we never allocate protected data and try to remember where it is with an unprotected pointer. On the next warm restart, we'll lose the pointer. Of course, if we reverse this (using protected pointer to unprotected block of memory), we'll end up with a dangling pointer. This at least can be fixed, by allocating a new unprotected block, but we have to remember to do so.

1.3.4.3 Cold restart

This is part of our normal software install sequence, although it may also happen if the hardware watchdog expires, the CP card is reseated, or its manual reload button is pressed. SYSLOAD reloads the database (and on non-flash machines, the code too), usually from a hard disk. Unsaved database changes are lost, and all calls are dropped. We then run roughly the same code as we would for a warm restart, but without the call-reconstruction phase.

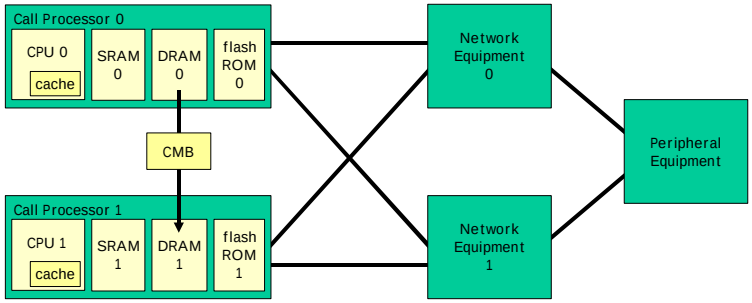
A cold restart takes roughly as long as a warm one, but has a more severe impact on the customer. Calls are taken down, and any unsaved database changes are lost. On the other hand, it almost always clears any memory corruptions.

If this cold restart is happening because the power has just been turned on, then there is one extra step. In this case, the very first thing we do before restarting the operating system is to write an "uninitialized store" pattern across all RAM, to set up memory parity. The reason this was important was that when we

ported to Thor, we discovered that several pieces of code were reading from memory *before* writing, which (apart from yielding meaningless results) would kill the restart. Since debugging a switch that won't come up is horribly difficult, this workaround was put in place.

1.3.5 Dual CPUs

1.3.5.1 Hot standby



Larger M1 systems are shipped with dual Call Processor cards. Either CPU is powerful enough to drive the whole system, and each is connected to all peripherals. The backup CPU is held in reset state, and is not actually running. The inactive DRAM bank is kept in sync with the active one by the CMB ASIC.

1.3.5.2 Graceful switchover

The hot-standby CPU is normally stopped. Therefore "you don't know if it works". Our customers actually invoke graceful switchover periodically (some are rumored to do it all day long, although once a day might be more typical) to ensure that the redundant cable paths and CPU are all still okay, just in case they need to do a real one.

Graceful switchover also happens under minor failures, if it appears that the CPU is healthy but the related components are not, such as IOP/IOC faults, CMDU faults, and memory parity faults.

Since graceful switchovers happen at a time when the switch is essentially healthy, it is reasonable to take a bit of time to prepare first, but the actual out-of-service hit to the switch must be minimized. On M1, there is currently a pause of about ¼second, and no calls are dropped.

The text segment (code) lives in flash ROM, which is duplicated, so you don't have to copy it. The patches do have to be re-applied on the new side. The data segment, both protected and unprotected, is mostly sitting in the reflected DRAM, so it's also available on the new side. The SL-1 stack, MMU context, exception vector table, and interrupt stack are in SRAM, and need to be copied across before resuming processing.

To switch over, we mask out all interrupts[3] to freeze the state, copy the stack and register state, reinitialize the devices (preserving IP addresses and physical device states as much as possible), and effectively do what looks like a return from interrupt to get things going again.

The "reflective-memory" system makes switchover fast (the outage is about a second), but we don't really want to build another one because it interferes with porting the OS and the performance of the system. Various vendors[4] are helping us look at supporting reflective memory without fancy duplex buses, using off-the-shelf hot-swappable compact PCI boards, and then providing automatic failover.

1.3.5.3 Ungraceful switchover (failover)

If the hardware watchdog expires, we suspect hardware problems and force a switchover to the backup CPU. There's no point spending time getting the other side ready; just switch CPUs and get back in service as fast as possible. Since we know there's some kind of fault in this circumstance, we do a warm restart on the new side in the hopes of clearing it. On M1, we currently take about 30 seconds to do an Ungraceful Switchover, and all calls that aren't in a talking state are dropped.

1.3.5.4 Split mode

In the lab, we allow redundant machines to be run split to double the number of test environments available to designers. This does not affect most application level software, but only one side gets to use the networks.

---

[3] Our flag should have been a readers/writers semaphore, but we never managed to get that going, so it's a straight semaphore.

[4] At least Chorus, Tandem, and Sun, and probably others before we're done…

1.3.5.5 Software delivery

We also use split mode to deliver new software loads to the field. The basic idea is that we split the switch, load the new software onto the inactive side, and do an ungraceful switchover to the new side.

In the primordial SL-1 software release process, the Integration Control Team used Overlay 43 (Datadump) to write the database out onto a tape containing the new software release. Next, they would change the jumpers on the Mass Storage Interface (MSI) board to tell it to boot from tape (or later, floppy) instead of from the hard drive. They would then reset the standby side, booting from this new tape. As part of the boot process, SYSLOAD converted the customer data to the new format where necessary. Once the standby side was ready to handle calls, an ungraceful switchover would be initiated, and after a brief outage the switch would start processing calls with the new software. The previously-active side could then be loaded from tape and made ready to act as the new standby. This process was one-way (you couldn't convert the new database into the old format if you wanted to back out later) but you always had the option of rebooting to a previously saved backup image if things went badly wrong. Also, until you had brought the old side back to hot-standby mode, you could cut back to the old load in a hurry if things looked bad on the new side.

There is still a vestigial tape emulation system (some like to call it an "abstraction") at the heart of the evolved software delivery system even though we haven't had any real tapes for years. And there is still a datadump "overlay", even though it is always resident in RAM. In CP1, we've shipped on floppies, and used a utility to load the new software onto the hard disk, and only ever booted from there. Since CP2, the image we normally boot from is in Flash memory. In the near future, most customers will probably have new loads and patches shipped to them over the Internet.

On a single-CPU system (Options 11, 21A, 21, 21E or 51), the process is similar, but you get a total outage during the reboot/rebuild process.

1.3.6 Packaging

With Thor (in particular with Option 11C), we started shipping the whole software load to all sites. Prior to that, unpurchased software was optimized out of the load, and you had to get a new cartridge if you wanted to install another package. Now everything except mobility is there, but some is deactivated. If you purchase a new feature, we tell you a magic "key code" which you can use to

enable the software. This means switches need a bit more memory, but customers don't need to wait for us to ship them new packaged loads.

1.3.7 Tools

1.3.7.1 Patching

If serious bugs are detected in a load which has already been shipped, we need a way to fix the installed loads, sometimes with some urgency. Historically, we had a very narrow bandwidth to switches in the field (especially outside of North America), so we couldn't afford to send a large chunk of object code each time (and in particular we couldn't just send a new image file). In any case, we would usually be dealing with a customer who was already unhappy, so we couldn't disrupt their system further just to fix the problem. We didn't want the cure to seem worse than the disease.

The required magic is performed roughly as follows. We figure out the required software update using standard lab tools. We compile and link the module(s), producing a new load file. Using a variety of tools, this gets built into a patch file, which is transmitted down to the site. There the patch is loaded into RAM, and the MMU is told to map calls to the old procedure (usually in flash ROM) to the new location. Thus we can alter the software with no interruption to call processing, even on a single-CPU system.

The downsides are that the patched code is more error prone (because it can't be tested as thoroughly and is often written under time pressure); that it runs more slowly (because it is not in flash); that it must be re-applied on cold restarts and switchovers; and that there is some extra administrative overhead. All of this means that we want to keep the total number of patches small.

T00116, the *Thor Patcher Users Guide*, is in Doctool library TOOLDOCS or on the web at http://47.82.33.147/~raviyer/TOC.htm. The Meridian Patch Library reference guide is on the web at http://47.58.130.173/BIG/MPL/mpl.pdf.

1.3.7.2 Problem Determination Toolkit (PDT)

PDT is the low-level debugging utility built by the Thor team. There are a number of existing documents describing how to use it.

Try F02824, *Thor Lab User's Guide* and F03226, *Thor Technical Notes* in Doctool library MLVDOCS, or the Debugging Techniques chapter of *Inside the*

Option 11C at http://47.75.6.2:8080/common_cts_info/PDF_Files/Inside_Opt11C.pdf.

1.3.7.3 sl1Spy, sl1qShow, systat, memShow, segPctShow

These tools check on the overall health of the system, and are particularly useful for debugging platform issues.

Inside Thor has a reasonable description of these tools, and they also have on-line help.

1.4 Maintenance frameworks

On top of the platform, or maybe surrounding the applications, we need a general purpose maintenance framework to help manage system. In the earliest days of SL-1, when the OS was not separated out, this framework was impossible to think about in any isolated way.

With the Thor project, we not only added the VxWorks OS, but the first generation maintenance framework, called the Hardware Infrastructure (HI). This is discussed in more detail in the Management chapter.

The following (now cancelled) generation used OO technology to take this a step further, creating the System Infrastructure (SI), which is covered in the Meridian Evolution chapter.

1.5 Intrinsics

Intrinsics were originally sort of indexed assembler language subroutines that either had to be blindingly fast or had to do things (like access I/O mapped addresses) that were hard to code in SL-1. These days they are all normal C subroutines.

1.5.1 Hardware intrinsics

These define our interface to the real hardware, eg: `IOREAD`. They are defined in module `intr.c`. Because they isolate most code from the details of hardware implementation, they make the job of porting between platforms somewhat easier.

1.5.2 Software intrinsics

These are mostly code we knew would be called a lot, which needed to be extra speedy, eg: TNTRANS. Until Thor, all intrinsics lived in ROM, and ran a bit faster than the rest of SL-1. Now all software is in flash ROM, but the software intrinsics are still a bit faster by virtue of being written in carefully tuned C rather than SL-1. They are defined in module `swintr.c`.

1.6 Third-party extensions to the platform

In Release 22, we stopped supporting the old bit-slice processors, and this allowed us to start experimenting with third-party software. Most of this ends up living between the previously existing operating system and the application code. The major pieces are:

- **Orbix:** an Object Request Broker from Iona, used for Mobility.
- **Seaweed:** a memory management system to keep Orbix from fragmenting the RAM. It overloads standard functions like `malloc` with its own versions (see Memory Manager in Chapter 3) so you don't have to do anything special to take advantage of its improvements.
- **RogueWave:** a broad spare-parts library of C++ classes, mostly for MAT, although Meridian Evolution would have used it too. See the book *Tools.h++, Foundation Class Library for C++ Programming* from RogueWave.
- **Envoy:** a Simple Network Management Protocol (SNMP) stack from Epilogue Technology, used mostly for MAT.
- **Retix:** an ASN.1 encoder used by Envoy.

1.7 Distributed processing

Intelligent Peripheral Equipment cards were developed to distribute more of the system's processing to its peripheral hardware. To allow for a flexible evolution of and ease of support for this distributed system, the ability to download microprocessor software into RAM for these cards is preferable to restricting software storage only to ROM. Existing cards which can do this are XNET, XPEC, and XNPD. The strategy calls for simple peripheral software, unaware of call state (eg: "dialing"), although it does understand terminal state (eg: "idle").

The peripheral is a slave to the feature logic in the core CPU, and its software is version-coupled with it.

Two new cards, MISP and MSDL, will follow the same path. One major enhancement that will be made for these new cards is the ability to store downloaded software in flash ROM on MISP/MSDL cards themselves. This will eliminate the requirement for downloading in the event of loss of power to the card, which will reduce the time required to bring up the cards to functional state.

We don't yet have any way of doing symmetric load-sharing between multiple CPUs, although we are discussing some possibilities.

1.8 Evolving the platform

The Call Server Evolution project is actively considering a major new platform, probably based on an Intel CPU, which among other differences is little-endian[5]. There is no generic way to write software that is both completely machine-independent and maximally efficient. As an application designer, what should you do?

The vast majority of differences in the first generation of a porting exercise will be taken care of by the compiler. This is still a big deal, especially with respect to support tools like source-level debuggers and patchers, but most designers can ignore it. The OS differences are likely to be masked if you've been writing to POSIX all along, but there is a chance that there will be performance problems, and new bugs (or at least subtly different interpretations of standards) so thorough testing is a must. SL-1 should mask the bit/byte/word packing differences, except for two important cases. The first is that booting from an old database could be an issue. This will usually be covered because we save a text

[5] According to the hacker jargon file, this excellent term derives from Swift's "Gulliver's Travels" via the Danny Cohen's famous paper "On Holy Wars and a Plea for Peace" [USC/ISI IEN 137, April 1, 1980]. The Lilliputians, being very small, had correspondingly small political problems. The Big-Endian and Little-Endian parties debated over whether soft-boiled eggs should be opened at the big end or the little end. "Big-endian" now describes a computer architecture in which, within a given multi-byte numeric representation, the most significant byte has the lowest address (the word is stored "big-end-first"). Most processors, including the IBM 370 family, the PDP-10, the Motorola microprocessor families, and most of the various RISC designs, are big-endian. Intel chips are little-endian. There's a theory that mathematicians naturally think of the least significant bit as bit zero, and want it stored on the right, while engineers naturally count from the left starting at one. The main thing is to agree, and capture it in your design documents!

file, which will automatically be converted, so you only have the usual version-dependence problems to worry about. The second case is when the CP is messaging to other processors, and this will need explicit planning to get right.