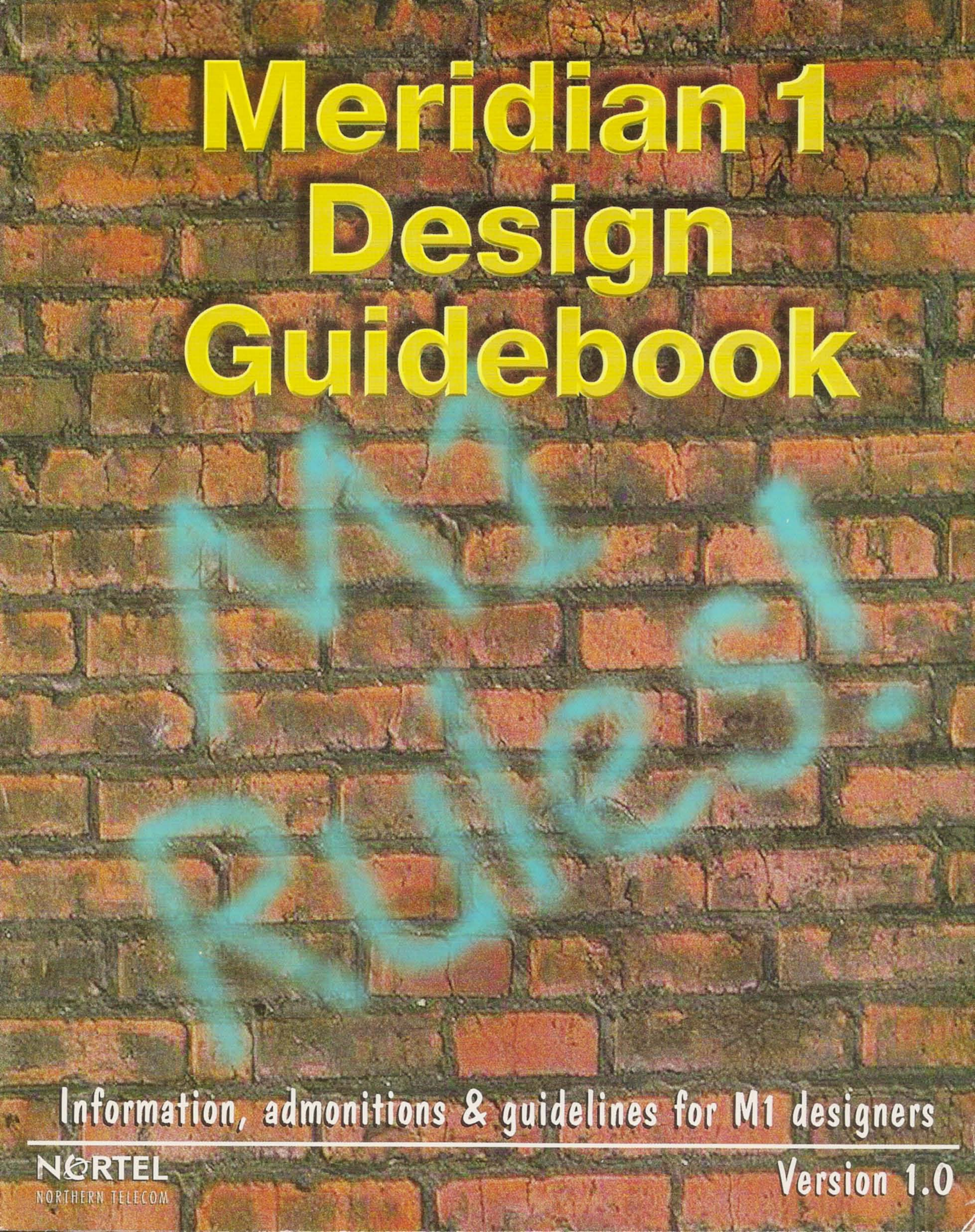
Meridian 1 Design Guidebook



Information, admonitions & guidelines for M1 designers

NORTEL
NORTHERN TELECOM

Version 1.0

Meridian 1 Design Guidebook

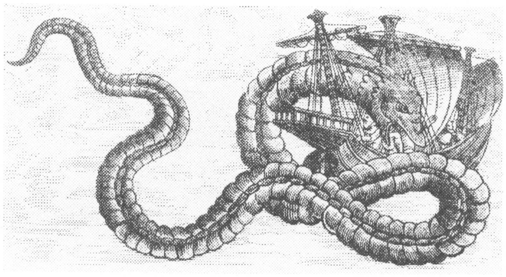
Information, admonitions & guidelines for M1 designers

Version 1.0
Issued July, 1998
Proprietary

NORTEL
NORTHERN TELECOM

Why does this book exist?

Meridian 1 software is a big, scary continent of code: millions of lines of source, multiple languages, decades of evolutionary change. New designers struggle to navigate software that's older than they are. Even old hands worry that the plate tectonics of recent projects have altered once-familiar territories beyond recognition. The most common travel advice today, from both managers and designers, seems to be "don't go there".

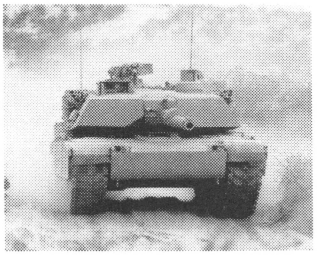

There be dragons here...

For a while, many folks hoped that we could simply seal off the SL-1 library frontiers and at least stay out of that jungle. Alas, fundamental changes, such as porting to a new CPU and operating system, adding wireless communications, or moving to an ATM switching fabric, mean that someone has to go explore how those changes will affect the system, and so expeditions continue to be mounted.

This book is an attempt to take some of the scariness out of such M1 exploration. Think of it as a travel guide: where to go, what to do when you get there, and what to watch out for. It maps out each of the broad areas of concern for an M1 designer. It tries to convey the essential patterns that recur in the system, so that when designers look at unfamiliar code (which is inevitable) they will understand what they see more easily. It points out the occasional minefield. It also explains some design rules and why they exist, which should mean that people find it easier to create better software. By understanding the issues, they'll even be able to make informed decisions *not* to follow the rules sometimes. Hyper-extending the metaphor, they can learn to be good M1 ecotourists, to avoid offending the natives, and to improve the chances that future visitors will have a good experience.

What wisdom this book contains has been distilled from the best advice we could elicit from various seasoned M1 travelers. Despite our best efforts to get our facts straight, it presumably still has some errors or omissions, and even the correct information will become out-of-date eventually. So, like all good travel guide publishers, we urge you to write back to us to tell us what recommendations were useful, what services have gone out of business, and what new things we need to include to cover the changing M1 geography. Tradition dictates that the best suggestions will be rewarded with a free copy of the next edition.

Our dream is that someday, if everyone learns and follows these rules, we may all be able to travel safely throughout the M1 world without one of these…



🡐 the M1 Tank
(no relation)

Happy trails,

[illegible]

Geoff Huenemann
(on behalf of the M1 System Architecture team)
July, 1998

BOGOSITY ALERT: The "M1 Rules" presented in this book are at best heuristics—they guide the process of discovering good designs by describing things that often work, but you still have to think through whether they're really right in any given circumstance.

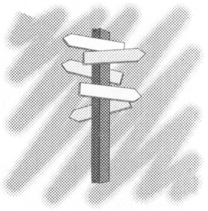
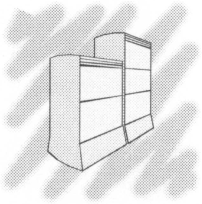


Table of Contents

Part I

Understanding the Context



1. A brief overview of M1

Before attempting to think about *how* a Meridian 1 does its job, we need to agree on *what* that job is. This chapter is the 30,000-foot view, providing the context for understanding the rest of the book. It will not tell you how to wire a BIX frame, nor what a circuit pack costs, nor how to fix that bug in call forwarding. But it will try to give a good overview of the markets we serve, the history of M1 development, the basic hardware components, and the neighboring systems with which we often interact.

Before proceeding, a word about nomenclature. In current marketingspeak, an SL-100 is a "Meridian 1 Option 201". However, although the SL-100 can host M1 peripheral equipment, it makes much more technical sense to think of it as a DMS-100 variant than as simply a bigger M1. Only the first three chapters of this book have any relevance to the SL-100. On the other hand, most of the information in this book has a fairly long half-life. It tends to apply to even the earliest SL-1 ancestors of the M1, and is likely to continue to apply for a few generations to come.

1.1 Market niches

Enterprises today can choose from three basic strategies for their phone service: PBX, Centrex, or VoIP.

Private Branch Exchanges (PBXs), like the M1, reside on the customer premises. They require a large initial investment in equipment and installation, plus an ongoing operations effort. The customer must allocate floorspace, provide a robust power source, plan data changes, coordinate software upgrades, and probably provide the first level of troubleshooting and a user helpline.

In contrast, Centralized Exchange (Centrex) business service is like outsourcing your PBX. It provides roughly the same services, but they are administered centrally by a local telephone operating company, and require no up-front capital. PBXs continue to win many sales over Centrex because of cost-effectiveness and the service responsiveness that comes from self-administration. But it's worth remembering that this competition is out there.

Voice-over-Internet-Protocol (VoIP) is the newest player in the game. IP is the *de facto* standard for data networking, and people are increasingly running something like 10 Mbps Ethernet to the desktop. In this context, even uncompressed 64 kbps voice starts to look inconsequential. However, there are substantial issues stemming from the optimization of IP for data: it's good at big packets and bursty traffic, and likes to retransmit in the face of errors. Voice needs lots of small packets with guaranteed limits on delays, and has no time for retransmission. VoIP has many potential forms, ranging from being a free long-distance trunk for your PBX, to being a fully distributed "unPBX". It has a bunch of disadvantages: it's less reliable; voice quality is uneven (for now); it adds to your LAN traffic; the standards are still emerging so you can't necessarily talk to other IP phones even on your own LAN; some of the "products" are still vaporware; propagation delay over the WAN can interfere with discussions; and it doesn't have most of the traditional PBX features (yet). But it does promise some slick computer-telephony features from good integration with your desktop computer, and ***it's really cheap***. If tariffing anomalies continue to make long-distance calls essentially free for VoIP end-users, then this will be our most serious competitive threat.

Despite all this competition, we've sold about 100,000 M1 systems worldwide. Here's why…

1.1.1 Classic Private Branch Exchanges

Most medium-sized organizations, such as corporate or government offices, universities, hospitals, hotels, and army bases, actively manage their own telecommunications. These groups tend to have a large number of phones. Their people would like to share a dialplan, probably need the same sorts of features, may share a balance sheet, and often call each other more than they call outside. Of the alternatives just mentioned, PBXs play very well into this market.

PBXs provide a richer collection of features than were historically available from a central office service. In fact, the *X11 Software Features Guide* is now 4½ inches thick, although each customer will normally buy and deploy only a small subset of the available features, depending on the nature of their enterprise. While most of these features are now available on Centrex too, regulatory and deployment issues mean that the fancy new ones will likely continue to be available on PBXs first. The following table gives a sampling of the kinds of features we provide to different clients.

Productivity & Time Management Speed call Hunting Call forwarding Ring again Last number redial ...plus zillions of others	**Cost Control** Line concentration Automatic route selection Call detail recording Direct inward system access Class-of-service restrictions Attendant administration
Applications – Horizontal Automatic call distribution Conference calling Multi-tenant service Voicemail Microcellular terminals Centralized attendant services	**Industry Segments – Vertical** International markets Finance Hospitality Manufacturing Health care Insurance

In practical terms, a "medium-sized organization" can mean anything from maybe 50 lines up to 10,000 lines. Smaller groups might choose a Key System, such as our Norstar product line which serves at most 272 users. Larger groups might pick something like our SL-100, which has sites today with over 40,000 users. A very important subset of our customers is in the small end of the M1 size range (<400 lines); the Option 11C accounts for nearly half of all lines sold, and great things are expected from the new 11C Compact, which only handles up to 116 lines.

A major distinction between our customers and regular telcos is that our groups use, rather than provide, telephone services. Therefore, billing requirements are much softer, and proprietary interfaces have been acceptable, although this is changing. However, because running the PBX is something many of these folks do in their spare time, administration ought to be easy. Also, the people in charge of providing voice communications within an enterprise are frequently different from those providing data communications. They tend to have competing budgets, and to have an over-simplified view of each other's job. Increasingly, each assumes their system could do both jobs adequately (Netheads claim voice-over-IP / Bellheads have ISDN), and each is still wrong. While we need to keep up with these developments, neither network has a very convincing story yet for sophisticated users of the other one, and the inertia of the installed base of wiring and equipment will keep both voice and data networks in buildings for some time to come.

To justify the initial cash outlay, people who buy PBXs expect them to be useful for a long time. They want to have cutting-edge services, but don't expect to have to toss out their older equipment to achieve this. One of our biggest challenges is to continue to support older systems while evolving the product.

A major recent addition to our basic PBX functionality is support for digital microcellular sets, allowing users to roam throughout the building speaking with the same digital handsets they use for the public cellular phone network. Now obviously, except in lead-lined buildings, they could have done this anyway; but with the M1 system, they use lower powered transmission, a building can support many more simultaneous calls, and best of all there are no air-time charges.

M1s are sold throughout the world. To be successful, they must be able to speak the local variants of standard trunk protocols. Also, the user interface of some features needs to be tunable to meet local expectations (from changing the ringing cadence in Europe, to having *katakana* set displays in Japan). Documentation and help screens must now be available in the "six core languages": French and German (for Europe); Spanish and Portuguese (for the Caribbean and Latin America); Japanese and Chinese (for Asia); and English. And of course, there are also many non-technical concerns that are equally imperative to our international success (politics, type-approval, currency, marketing, export and local legal issues, etc.) but these are outside the scope of this book.

1.1.2 Call centers

If Option 11Cs have the largest number of deployed M1 systems, call centers probably place the greatest demands on the architecture, and have the highest revenue potential. Call center customers tend to be very sophisticated telecom users. They quote their average call durations to three significant figures. They keep a very close eye on queuing times and maximum call throughput. They engineer their operations so that their operators' lines are almost constantly active. They care *very deeply* about outages. Most call centers are vital to the success of our customers' businesses, and a few are even used by emergency services bureaus. Therefore, call center managers would usually much rather spend money for reliability and speed, than risk losing client calls.

The cornerstone of call center service is **Automatic Call Distribution** (ACD), which allows any number of callers to dial the same directory number and be queued until a suitable operator is available. Conceptually, this is just first-in-first-out queuing, but it can get a bit convoluted. Based on Calling Line ID, the system might recognize some callers, and route them to a particular queue.

Other callers may have been given a special higher-priority number to dial. More than one service may be available, or services may be available in more than one language. Any given operator will be able to handle a different subset of these callers, and **Multiple Queue Assignment** (MQA) allows each to receive calls from the right subset of queues. Operators can also have their own individual directory numbers, which allow people to dial them directly.

Calls can be redirected, or given special announcements, based on the time of day, or their time waiting in the queue. Using **Customer Controlled Routing** (CCR), customers can write scripts to route a call through a series of announcements and queues.

Supervisors can eavesdrop on calls to ensure the quality of service is appropriate. With **Meridian MAX**, they can see real-time graphical displays of the length of the call queues, the time callers have been waiting, and statistics about how long operators are taking to handle calls. They can then reconfigure the queues in to optimize performance, or can generate reports to do long-term planning with the **MAXCaster** forecasting tool.

In some call centers, agents may actually sit in different physical locations from one day to the next, and the M1 must continue to send them the right set of calls and to associate them with the right supervisor. Using some third-party magic[1] we even support agents working from home.

Network ACD uses ISDN D-channel signaling to keep track of how busy up to 20 other sites are. It can then spill overflow calls intelligently to these sites using the standard ACD queuing software. Calls are queued locally until the off-site agent becomes idle, the agent is reserved, and then the call is delivered, optimizing network usage. Meridian **Network Administration Center** (NAC) allows centralized network control for organizations with multiple Meridian MAX sites. The NAC software collects information from Meridian MAX call centers and processes it in a single microcomputer, enabling the customer to monitor and control agent/queue performance at all nodes from one central point.

Integrated Voice Response (IVR) automates some of the process of routing an unknown caller to the right agent, and may even be able to perform simple query services independently. This allows companies to provide better 24-hour service, to handle spiky traffic patterns better, or to save money by having fewer operators.

[1] The Off-Premise Extender from MCK Communications – see www.mck.com.

Using **Meridian Link**, Computer-Telephony Integration (CTI) allows ACD calls to be integrated with a company's data processing system. For instance, when an agent is presented with a call, the **FastView** software could simultaneously present the customer's order history on her computer screen. Or, if the caller hasn't paid his bills, he might be routed directly to the Collections department. The "predictive dialing" feature of CTI automatically originates annoying telemarketing calls during quiet periods.

An extension some analysts expect to see soon would be to have call center operators fielding multimedia "calls", including ones that came from email, fax, internet surfers, and video terminals, in addition to voice calls. The underlying technology works today, although integrated, scalable, product-quality offerings are a bit thin on the ground. But a fundamental obstacle to deploying such technology is that not all people are good at communicating in all media, so it isn't clear how easy it would be to staff a multimedia call center.

Originally known as Integrated Call Center Manager, and then briefly as Next Generation Call Center, **Symposium** is the latest version of our ACD offering. The new configuration uses a Symposium Call Center Server (SCCS) to control all of the queuing off-board from the M1. SCCS uses scripts to control "Skill-Based Routing" and integrated IVR to optimize operator efficiency. Other than that, it doesn't do too much that's new, but it does cut down on the total number of boxe needed to do the same old (important) job.

1.1.3 Local points of presence

In most environments, telephones are not in constant use, and many calls connect people within the same building. Therefore, except for Call Centers, M1s tend to do significant concentration. That is, there are many more lines served by the M1 than there are trunks connecting to the outside world[2]. In a sense, the M1 is always a local "point of presence" of the global Public Switched Telephone Network (PSTN).

However, there are a number of situations in which customers think of M1 equipment specifically as a small local point of presence for a larger switch. Often, it is only the larger system that has any direct connection to the PSTN. While the difference between this way of deploying an M1 and the classic PBX is mostly in the mindset of the customer, the requirements and the economics may be different from other cases.

2 The author can attest to one extreme case, a six-story office building in India that had only *two* outside lines for general usage.

1.1.3.1 Remote peripherals

Carrier remotes

The usual way to handle a group of M1 lines located some distance from your switch is to use public carriers (T1 or E1) to connect remote peripheral equipment up to 70 miles from your M1. This can be especially useful when you have a number of small branch offices around a city. Each site will be centrally administered just as if all the lines were local. The downside is that you have to pay an on-going charge for the links.

Fiber remotes

Peripherals can normally be only 45 feet from the M1 that supports them. If you're lucky enough to have right-of-way between your users (say, between two buildings on the same campus) you can use fiber optic cables to connect remote peripheral equipment up to 3.1 miles from the main M1. These remote units then have the full functionality of a local peripheral.

Line-side T1

Sometimes the easiest way to wire a group of lines some distance from the main switch is to use a Line-Side T1 Interface (LTI). The LTI card plugs into 2 slots of an IPE cabinet, and promises to behave just like a normal analog line card (XALC). The difference is that, rather than terminating a bunch of analog lines, an LTI card has a full T1 (or E1) trunk coming out of it. Now, instead of running 24 individual wires out to your cluster of phones, you can run the T1 to a Norstar key system, or simply to a channel bank, which does very simple-minded de-multiplexing of the T1 channels back to analog lines. Besides making wiring easier and cheaper, this also extends the maximum distance the phones can be from your switch room to 655 feet (farther with repeaters, around the world if you lease a public T1).

There is arguably a more important deployment of LTI technology. Notice that the M1 *thinks* it's just got a bunch of lines out there. This means that all line features are automatically available (which would generally not be true with a real T1 trunk). Several third-party vendors have also noticed this, and there is a flourishing sub-industry of people building specialized boxes to do ACD, IVR, wireless, voicemail, etc., on the far end of one of these T1s. This method of connecting voice lines is cheaper, and also has better speech quality because it skips several passes of digital to analog voice conversion. In fact, because we are a global PBX market leader, it seems that any stable interface the M1 has provided, no matter how arcane, has been reverse-engineered by third-party vendors and reused to build new services. With the advent of TAPI and other public call control standards, we may find fewer small vendors as willing to do this.

LTI cards are actually built by a partner company, Telecom Strategies Inc., but they are fully supported by Nortel.

Sometimes the M1 *is* the remote peripheral

Some of our customers have deployed small M1s (especially Option 11s) as remotes to larger M1s. This gave them an extremely robust and feature-rich remote system, but at a higher cost and with more administrative hassle. Part of why this happened is that IPE carrier remotes were not available for many years.

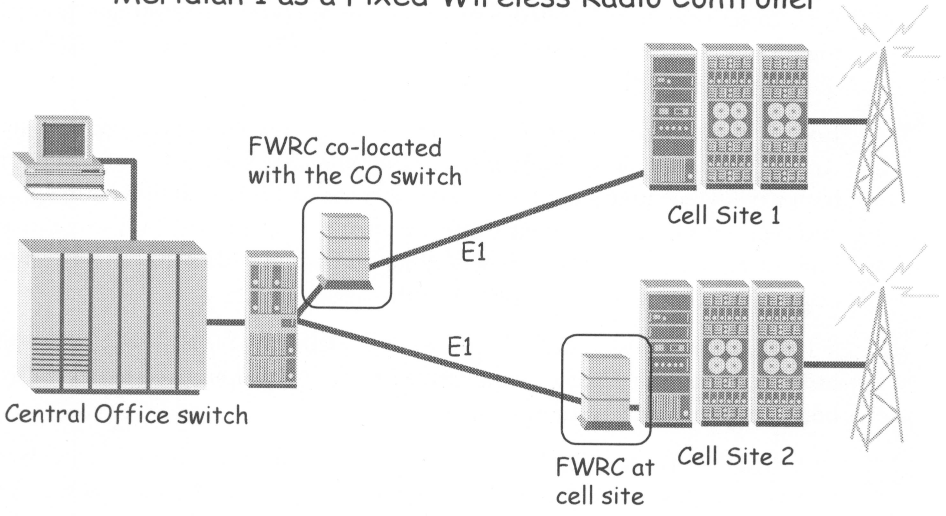
1.1.3.2 The Fixed Wireless Radio Controller (FWRC)

In many third-world markets, there is a huge demand for more phone services. It's normal in places to wait more than a year for a new line, and some rural areas have no service at all. But for a new provider to be able to survive financially, their cost per line must be minimized, and they must be able to deploy new networks quickly and realize revenue without investing too much in outside plant cabling or major switching fabric components. Using wireless telephony, operators can provide basic telephone service quickly, installing radio sites and towers—instead of trenching, cabling, and pole planting—and eliminating the need for many kilometers of copper wire. Radio sites are both easier and cheaper to maintain and operate than copper line plant; this translates into lower annual recurring costs for the network operator. The service provided is essentially identical to traditional wireline phones, but instead of being plugged into a jack that connects it through traditional twisted-pair copper wire to the local telephone switch, the telephone links to the network by means of a wireless radio path. New subscribers can be added very quickly, without sending technicians to the new site. The system is especially cost effective even in areas with very low subscriber density, or very rugged conditions.

Nortel offers a fixed wireless solution to this market called Proximity T, based on Time Division Multiple Access (TDMA) radio. A central office switch talks to a group of FWRCs, each of which is basically a Meridian 1 Option 51C with mobility software.

The FWRC performs the dynamic radio resource management function, assigning radio channels to calls as they are set up. In addition, the FWRC includes the cards which perform the IS-54 voice coding required for digital TDMA transmission. The FWRC can be located either at the Central Office or at the cell site. For added reliability, a pair of FWRCs may be deployed for each cellsite. The wide variety of trunk protocols supported by the M1 makes it particularly suitable for this market.

Meridian 1 as a Fixed Wireless Radio Controller



1.1.3.3 The Multimedia Carrier Switch (MMCS)

Marne-la-Vallée has recently announced an Option 81 variant optimized as a market-entry switch for international public carriers. Even five years ago, nearly all national PSTNs were run by a government department called a Post, Telephone and Telegraph (PTT) administration, often a branch of the post office, which sourced its equipment from a highly-protected domestic manufacturer. Alcatel, Lucent, Ericsson, Siemens, NEC, and of course Nortel, all grew fat on this system. The recent global trend toward at least privatization, and usually deregulation, has created a nice big surge in an otherwise pretty flat (or completely closed) public switching market. In 1998, most of Europe opened up the cross-border telecom market to competition, and a number of "supercarrier" alliances formed to capitalize on this opportunity. On a smaller scale, regional carriers and Alternative Licensed Operators saw similar opportunities and faced similar issues. North American Competitive Local Exchange Carriers (CLECs) are starting to explore a similar niche.

One basic problem they all face is that even though their initial traffic loads will be light, they need to have a lot of switching points in order to offer a credible service. They can't afford to put a full-blown DMS into every country. The

MMCS gives them a cheaper alternative. The theory is that once we've captured a carrier's attention, we will likely retain it even across a non-transparent upgrade (like M1 to DMS). Notice that at one level the MMCS, like the FWRC, is just a protocol converter to give service providers a local point of presence.

A second problem is that to connect with each country's existing infrastructure, the local switches will have to support the local protocol variants. M1 already does this fairly well.

Enhanced services (Calling Card, Debit Cards, Freephone, Callback, Equal Access, Operator Services, Intelligent Networking, Virtual Private Networking, better billing accuracy, etc.) increase the value of these switches to the alliances and their end users. Some of these are already available on M1; some need to be built or enhanced to meet the new requirements. In MMCS, the default will be to build them on off-board Application Processors.

While work is being done to improve the reliability of the MMCS, it is still at heart not a Central Office-compliant architecture. This could conceivably become an obstacle to deployment.

In the initial offering, the "Multimedia" part of the name is pretty much entirely hype. The basic switching and access connect standard 64-kbps voice. Yes, this gives you fax and modem data for free, and video for only a bit more work, but it hardly gives you bragging rights… Expect some work in the immediate future to tackle this, perhaps with the MMCS ending up as a value-added gateway to the Internet.

Recent marketing carefully positions the MMCS node as "a Core Switch, *based on* M1 technology", rather than simply an M1. This seems to be in order to emphasize the AIN-style call control model, and possibly to allow higher margins to be negotiated ☺.

MMCS seems to have a very high standard of on-line documentation, at least some of which ought to be leveraged by the broader M1 community. Curious readers might start their investigations at http://47.74.128.167/mmcs.html.

1.1.4 Virtual Private Networking (VPN)

The recent flurry of international business mergers and expansions has accelerated the demand for flexible, cost-effective, manageable, private networking. *Virtual* Private Networking combines the administrative convenience and security of a closed private network with the ubiquitous reach and cost-effectiveness of public trunks for long-haul connections. A little more than half of our switches go to customers linking multiple sites into a VPN.

In general, M1 tries to coordinate VPN services with SL-100/DMS-100 and DMS-250, under the marketing banner of **Meridian Customer Defined Networking** (MCDN).

1.1.4.1 Electronic Switched Network (ESN)

ESN is a broad collection of features that allows customers to build a VPN. It includes both coordinated databases (for uniform billing and dial plans) and telephony feature networking based on ISDN PRI trunks.

ESN lets customers optimize costs by laying their own cables where geography and legislation permit, choosing leased lines between some strategically-chosen sites, and routing over the public network between other sites. Overflowing calls can be re-routed internally or externally, or can be blocked. This can end up saving a lot of money.

ESN allows a uniform behavior across all corporate sites (as long as we can agree with DMS about feature look-and-feel). ESN provides network-wide caller name and number display, Networked Attendant Services and ACD, and miscellaneous networked features like Ring Again and Call Pick-Up.

Depending upon national regulations, callers may even be able to dial from home into the ESN using Direct Inward System Access (DISA) and/or break out to the PSTN at the terminating end to bypass long-distance charges.

Mastering the full range of ESN capabilities is a somewhat Herculean task, but answers to specific questions may be found in the ESN NTP, available in printed form, on CD-ROM through Helmsman, or at http://47.5.1.11:8000/data/Knowledge_2/Switching/MMCS/NTPs/07_ESN.PDF.

1.1.4.2 Virtual Network Services (VNS)

VNS provides VPN services to smaller locations that cannot justify dedicated ISDN trunks, or can only accommodate a limited number of them. VNS is based on a dedicated D-channel to do the network messaging, but can use most PSTN or Tie trunk types to carry the associated voice. People familiar with the CCS7 public network model will recognize that this is a similar idea on a smaller scale.

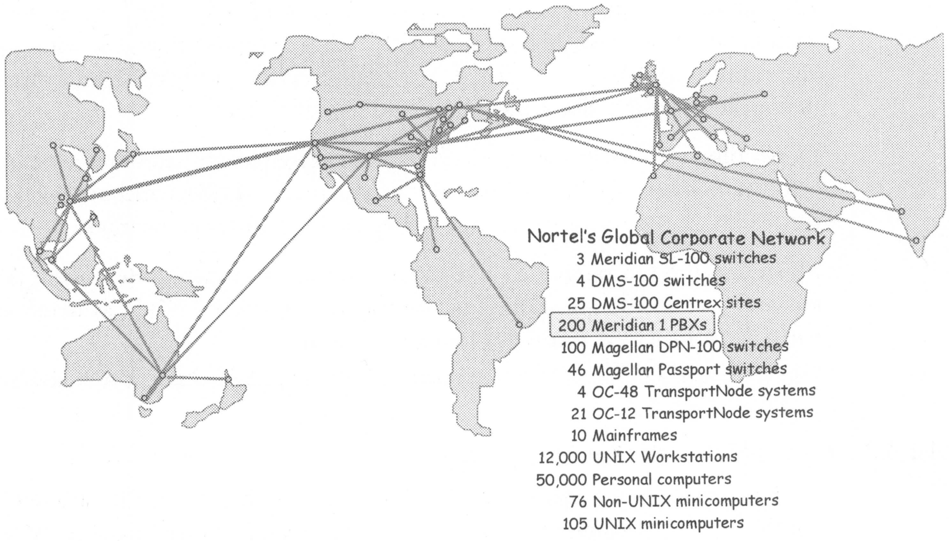
VNS supports such MCDN features as trunk anti-tromboning, Network Call ID, and Network Call Page, NAS, and NACD.

1.1.4.3 In-building VPNs

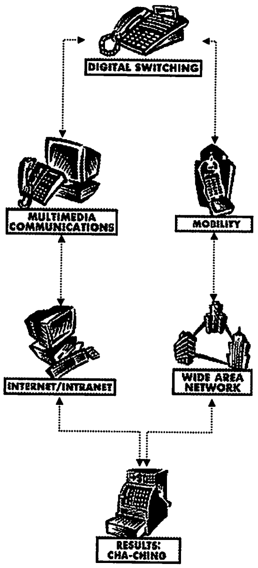
A special case of ESN is for a customer who wants to buy a bigger or more disaster-proof M1 than we know how to build at the time. There have been instances when we have networked a pair of M1s together in the same building to do the job of a single switch. (If you were wondering who would ever buy the Networked Call Pick-Up mentioned above, this is your answer.) There is enough overlap between the high end of the Meridian 1 range and the low end of the SL-100s that this pattern may stop happening in the future, although theoretically you might sell some twin configurations to people who were very worried about disaster survivability.

1.1.4.4 CorWAN

An obvious VPN example to consider is Nortel's Corporate Wide Area Network, our own 60,000-user ESN, which turns out to be one of the most advanced private telecom networks in the world, and uses M1s as part of the core technology. In 1997, it looked like this:


Nortel's Global Corporate Network
3 Meridian SL-100 switches
4 DMS-100 switches
25 DMS-100 Centrex sites
200 Meridian 1 PBXs
100 Magellan DPN-100 switches
46 Magellan Passport switches
4 OC-48 TransportNode systems
21 OC-12 TransportNode systems
10 Mainframes
12,000 UNIX Workstations
50,000 Personal computers
76 Non-UNIX minicomputers
105 UNIX minicomputers

1.1.5 Power networks



They're the latest thing. As of last year, more of the world's network traffic is data than voice. And data is growing in some markets at roughly 100% per year, compared to voice's 15%. Even our 15% growth is being driven by fax, video, internet access, and messaging (all of which are pretty data-ish). Running parallel networks is expensive. This should mean that instead of having voice networks with data overlays, the world will increasingly have data networks with voice and multimedia overlays.

Recent propaganda claims that NORTEL POWER NETWORKS™ are well positioned to help enterprises deal with this: Power Networks are "responsive, reliable, simple, secure and flexible". The central thrusts are to extol the merits of switching over routing (although "data" equipment is increasingly able to do this too) and to leverage our history of good reliability. It's not entirely clear what this means to M1, but presumably an excellent Power Network solution would have leveraged the ATM backbone in a Meridian Evolution switch—once we had a good native-ATM port story. Whatever we do instead will have to play well into this new marketplace. And presumably, this road leads to "Webtone".

1.2 Customer values

The different market segments just discussed each emphasize different variables, but overall the things our customers would call "quality" in a PBX are not too surprising.

To start with, they're all the things people have always liked about the phone system. The themes which spring to mind are extreme reliability; low cost-of-ownership, with seamless growth at incremental cost; simplicity of installation, operation, and end-customer use; near-zero transmission latency; suitable management and billing systems; a rich feature set; support for global protocol standards; and secure access and transmission.

Then there are some newer things we do, like micro-cellular, Computer/Telephony Integration (CTI), and craftsperson interfaces in local languages.

The tougher part is that, added to all of these values, our customers will increasingly look for the frequently-conflicting values they've come to expect from the computer industry. They will want open interfaces (implying an infinite number of deployable configurations to test), and Graphical User Interfaces (GUIs) for system management. They will want to be able to run software written by a variety of groups, including themselves. They'll want high payload bandwidth with flexible access choices, probably including IP over Ethernet. They might demand that we double our computing power every eighteen months, like the nearly-disposable PC market does, without weakening our evergreen promise! They'll also be comparing our carrier products with the "free" internet equivalents. Some of these requirements pull us in opposite directions. There are some challenges here.

Finally, "quality" should include some important non-functional properties that are visible mostly inside of Nortel. They affect the customer only indirectly, but are nonetheless critical if our designs are to have any long-term value that could help us maintain our competitive position. Included in this list are things like maintainability, flexibility, testability, portability, and reusability. Our success with all of these "-ilities" predetermines our agility in an evolving marketplace, and ultimately our success as a business.

How to tell if your customers value something...

1.2.1 Size matters

One of our strengths in the market is that we offer a line of products which scale from a small site (maybe 50 phones) up to a fairly large one (several thousand). A company which starts small can gradually add extra users and features, leveraging most or all of its initial investment.

The hitch is that small customers are *qualitatively* different from large ones. Small customers will want to add "manage the PBX" to the list of regular duties of a secretary or computer support person. They need a system that is easy to set up and maintain, and good support when they can't figure it out. They are very cost conscious, and probably can live without redundancy. In contrast, large customers tend to be "power users", with dedicated telecom personnel who know as much about running a PBX as we do. They stretch the limits of capacity and feature interactions, and will probably have some non-Nortel equipment that they expect us to interact with nicely. They tend to have networks of PBXs which include remote offices (bringing us back to the needs of small sites).

Our ability to evolve a switch from the small end to the large is a distinct marketing advantage, but also we need to continue to ensure that our products make sense for the various niches in between. This is a potential weakness with a closed solution like the Option 11C Compact.

1.2.2 How do we stack up against the competition?

Bearing in mind the diverse set of requirements discussed above, how easy do our users find it to make a Meridian 1 fit their needs? M1 has led sales in North America for several years now, currently capturing about a third of all lines sold, so we must be getting something right. The market changes rapidly, but numbers below giving our rank in 1996 in various measures of usability, compared to the competition, are probably still representative. There's certainly room to improve (we're not number 1 in any category), but there are no huge weaknesses either, and we ranked second place overall. Soon, companies like Cisco (LANs) and GroupComm (unPBXs) will have to start showing up in such comparisons...

Usability Criteria	Mitel SX-200	Nortel M1	Ericsson MD110	GPT iSDX	Lucent Definity G3	Siemens Hicom 300	Alcatel 4000 Series
Ease of Installation	1	4	3	2	5	6	7
Ease of operation (attendant)	2	4	2	1	6	5	7
Ease of operation (station users)	1	3	2	4	6	7	5
Maintenance	1	4	3	2	5	6	7
Moves, adds & changes capabilities	1	3	2	4	5	6	7
Price/performance ratio	3	2	4	1	5	7	6
Service and support	4	2	5	1	3	6	6
Software upgrades	6	2	1	7	4	3	5
System documentation	1	3	2	5	4	6	7
System management	1	2	5	4	2	6	7
Training programs and materials	5	3	2	4	1	6	7
Overall Usability *(1=customer favorite)*	1	2	3	4	5	6	7

1.3 Distribution channels

Unlike many Nortel products, Meridian 1 systems are usually sold and supported by third-party distributors, rather than directly by Nortel. These distributors include telecom companies like WilTel and Sprint, divisions of big diversified corporations like Daewoo and Xerox, some regional Nortel subsidiaries, and the occasional national PTT. They work together with us through regional Distributor Alliance Councils (DACs). Distributors provide:

- service centers throughout their territory
- 24 hour × 7 day a week emergency support
- field engineers trained by Nortel
- remote or on-site problem diagnosis and resolution of customer-reported problems
- spares for emergency hardware replacement
- installation, project management, traffic/security audits, customer training

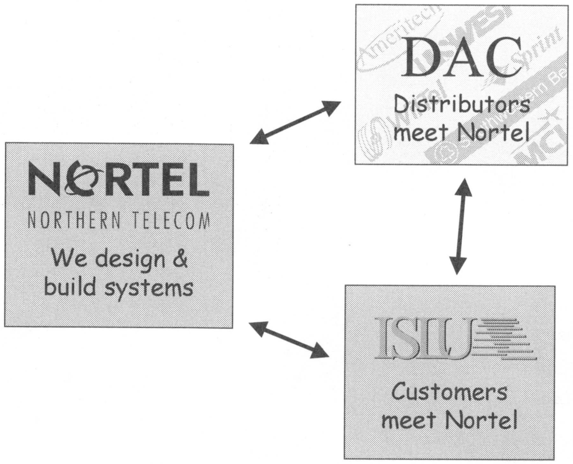
Various groups within Nortel back up the distributors with training, emergency assistance, project management, audits, and cutover assistance.

Outsourcing of both sales and support puts a level of indirection between the customer opinions charted above and the design community. While the distributors work hard to maintain their technical skill, we can help them out by making our systems as self-configuring, self-diagnosing, and self-healing as possible, and by providing excellent documentation. For more information, the Asia-Pacific DAC has a good web site at http://47.75.6.2:8080/members/index.html.

Also, while it is in the best interests of the DAC to convey customers' wish-lists to us, the extra step could mean we lose touch with current priorities. One of the ways our customers get at us directly is the International SL-1 Users' Association (ISLUA) Product Advisory Committee (see http://47.245.71.104/islua/ or http://www.islua.org/). The ISLUA was established in 1980 and has over 5000 members. They hold a big annual 5-day conference where they attend educational workshops, see demos, swap whispered tips on gray market[3] sources,

[3] One potential problem with building equipment that lasts a long time is that it may outlast its usefulness to any given customer. These customers will, understandably, want to resell it, but this presents a conundrum for our distributors: they lose a sales opportunity and they may inherit a support cost, but at the same time their customers probably like a healthy gray market. The gray market also messes up our pricing strategies, which are based on what the market will bear rather than our direct costs for any given piece of equipment or software. We'd like to (and do) charge more in markets that will pay more, but the gray market (here some would substitute the term "free enterprise") limits our ability to do this. Corporate Security figures Nortel concedes about $8 million a year in profits to the gray market, although for obvious reasons an exact figure is hard to come by.

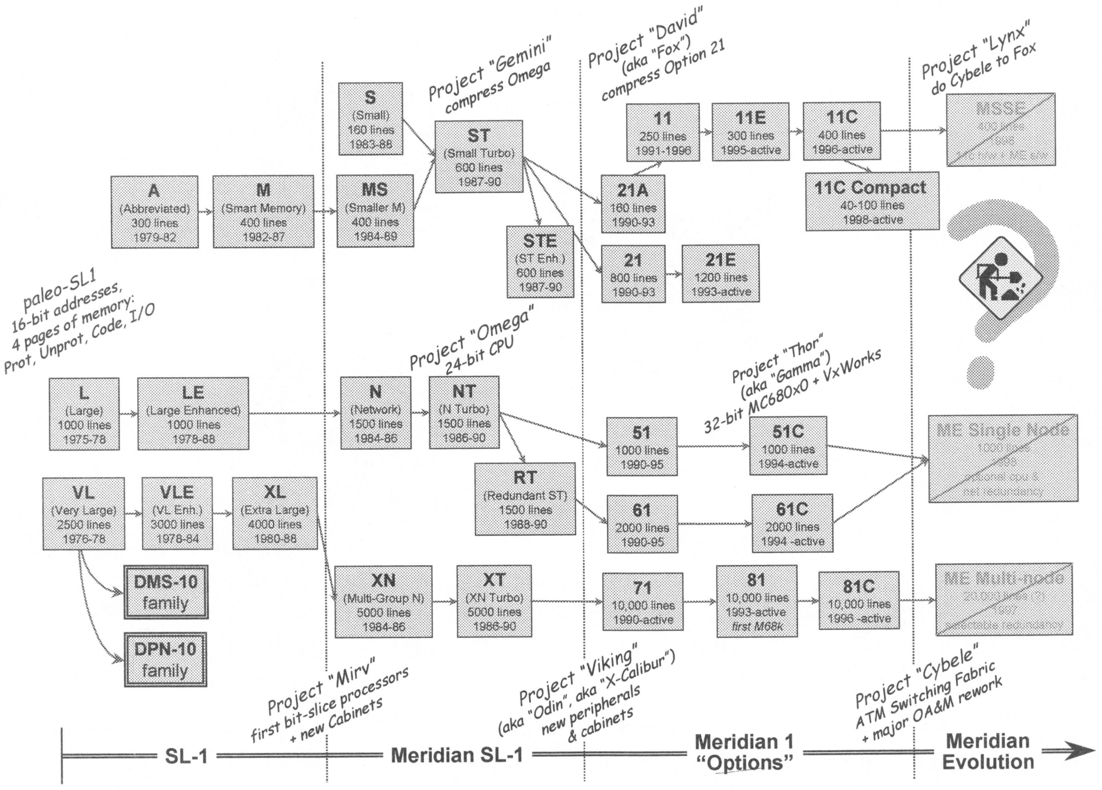
and tell us to our face what they think of our stuff. Of course, the ISLUA is inherently bad at telling us why people have chosen *not* to buy our products, but at least it can tell us a lot about our existing customers.



Keeping the distributor channels healthy is obviously critical, but we should resist the temptation to optimize only for distributors rather than real customers. For example, a cryptic user interface that requires highly-trained operators might inconvenience customers but would nourish a lucrative consulting business for the DAC members…

There's actually a dilemma inherent in our business strategy. Anybody's product needs to be profitable for its supplier, and valuable to its customers. Ours must also make money for the distributors. Even if Nortel could make just as big a profit by building our systems for a tenth of the price, and charging our end users a tenth of the price, it might still drive our distributors out of business. As the unPBX folks change our industry, we're going to have to grapple with this issue.

1.4 M1 genealogy


paleo-SL1
16-bit addresses,
4 pages of memory:
Prot, Unprot, Code, I/O
A (Abbreviated) 300 lines 1979-82
M (Smart Memory) 400 lines 1982-87
S (Small) 160 lines 1983-88
MS (Smaller M) 400 lines 1984-89
Project "Gemini" compress Omega
ST (Small Turbo) 600 lines 1987-90
STE (ST Enh.) 600 lines 1987-90
Project "David" (aka "Fox") compress Option 21
11 250 lines 1991-1996
11E 300 lines 1995-active
11C 400 lines 1996-active
11C Compact 40-100 lines 1998-active
21A 160 lines 1990-93
21 800 lines 1990-93
21E 1200 lines 1993-active
Project "Lynx" do Cybele to Fox
MSSE
L (Large) 1000 lines 1975-78
LE (Large Enhanced) 1000 lines 1978-88
Project "Omega" 24-bit CPU
N (Network) 1500 lines 1984-86
NT (N Turbo) 1500 lines 1986-90
RT (Redundant ST) 1500 lines 1988-90
Project "Thor" (aka "Gamma") 32-bit MC680x0 + VxWorks
51 1000 lines 1990-95
51C 1000 lines 1994-active
61 2000 lines 1990-95
61C 2000 lines 1994 -active
ME Single Node
VL (Very Large) 2500 lines 1976-78
VLE (VL Enh.) 3000 lines 1978-84
XL (Extra Large) 4000 lines 1980-88
DMS-10 family
DPN-10 family
XN (Multi-Group N) 5000 lines 1984-86
XT (XN Turbo) 5000 lines 1986-90
71 10,000 lines 1990-active
81 10,000 lines 1993-active first M68k
81C 10,000 lines 1996 -active
ME Multi-node
Project "Mirv" first bit-slice processors + new Cabinets
Project "Viking" (aka "Odin", aka "X-Calibur") new peripherals & cabinets
Project "Cybele" ATM Switching Fabric + major OA&M rework
SL-1
Meridian SL-1
Meridian 1 "Options"
Meridian Evolution

In 1876, Alexander Graham Bell built the first telephone. A century passed, and the underlying technology remained more or less unchanged, although the world got progressively better at building electromechanical switches. In the late 1960s, various people started using very simple computers to control transistors switching analog voice, and the first analog PBXs appeared (including the "Stored Program 1", from a company called Northern Electric).

Then in 1972, that company (soon to be renamed Northern Telecom) started work on the world's first PBX to be based on time-division multiplexing of digital voice: the "Stored Logic 1". 1975 saw the first field trials (in Belleville and Montreal), complete with the "SL-1 set": analog voice, digital signaling, hands-free capability, 13 function keys, and no display. For comparison, around this time Zilog started shipping the first generation of Z80s. (Remember the TRS-80?)

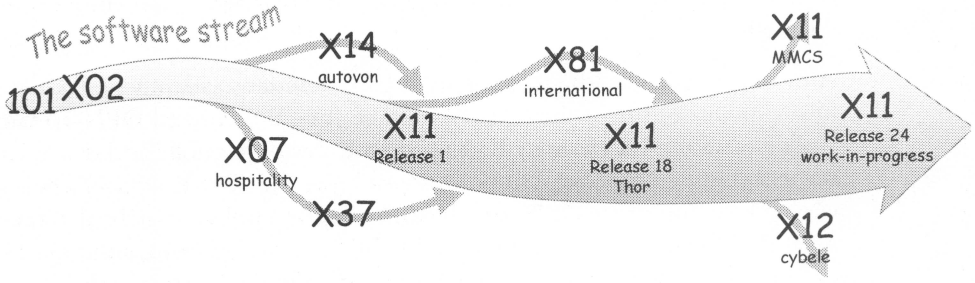
Almost immediately, the technology began to undergo adaptive radiation to play into various sectors of the telecom market; the DMS-10 and DPN-10 families forked off from the original SL-1 product to become public and data switches, respectively. To this day, DMS-10 is written in the SL-1 language, albeit an evolved version of it. DMS-100, Nortel's flagship public switch, also reused a great deal of the knowledge built up by the SL-1 design team, although they opted to redesign most of the hardware and software from scratch.

Over the years, the SL-1 got progressively faster and smaller, and a variety of packaging strategies targeted different sizes of customers. In 1980, AMD built us a speedy new bit-slice processor to power the SL-1 XL up to 4000 lines. In 1983, the "Mirv" project doubled the CPU capacity while new cabinet work shrunk the footprint to create the "Meridian SL-1" line. Two years later, we moved to the "Omega" CPU: a massive 24-bit architecture (well, it seemed big at the time). In 1988, "Viking" re-worked the networks, peripherals, and cabinets, and defined an upgrade strategy all the way to SL-100, to give us the "Meridian 1" line. In 1990, the "David" project (as opposed to Goliath) cost-reduced the same architecture (4 × 4-bit AMD chips) onto a single gate array to bring us the "N07" ASIC for Option 11.

Then in 1990 came the big push for a commercial platform (the "C" in "Option xxC"). "Thor" moved us to a MC680x0 CPU. Since then, we've been able to ride Motorola's commercial processor curve fairly well, replacing the MC68030 with later generations of CPUs and memory cards without substantially reworking the bulk of the system.

It's probably worth remembering that many of the older generations are still out there (the oft-lamented "stalled base") and they ought to be a good market for upgrade sales. The problem is that the equipment was built to last, and for those customers whose needs really haven't changed, the old machines continue to perform their jobs fairly well. To sell new machines, we have to show clear benefits at sensible prices, and have a good upgrade story. People want to reuse as much equipment as possible, and outages must be kept to a minimum.

Throughout this evolution of the hardware, the software was also developing. Each release built on previous ones, and added many new features. The first software stream was called 101. The next release supported two machine types, so we called it X02: 102 for the L, and 202 for the VL. We continued with X03, X04, and X05, until X11, at which point we changed the naming convention to be X11 Release 1, X11 Release 2, …



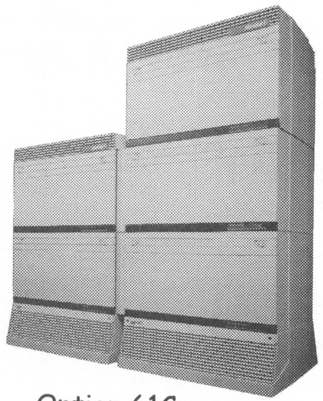
New projects are often tempted to go off on tangential side streams to allow them to release features as early and often as they wish, but then designers need to do the work of keeping current with advances in the base manually. X07 was a special release for the Hospitality market, and developed further as the X37 side stream before being reintegrated onto the main steam in about X11 Release 12. X14 was a special Autovon stream, later reintegrated in X11 Release 3. X81 had the code for the international market, and was reintegrated in X11 Release 20. MMCS didn't actually take different generic or release names, but issued sub-releases on top of standard X11 releases.

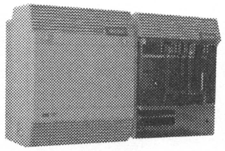
X11 Release 18 put SL-1 on top of VxWorks. Release 21 was the last generic that had to support the older bit-slice CPUs, which allowed Release 22 to go crazy. We added a raft of third-party code, mobility, MAT, and SCCS. X12, the first new generic in years, did the major rework for Cybele, particularly on the connection service model, and created the System Infrastructure.

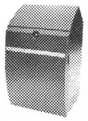
1.5 The hardware

This section attempts to be representative, rather than exhaustive. For a fairly complete set of current Nortel products, check out http://www.nortel.com/home/quick/.

1.5.1 Physical packaging


Option 61C
(5 UEMs in 2 columns)


Option 11C
(shown with one expansion cabinet, and one front panel removed)


Option 11C Compact

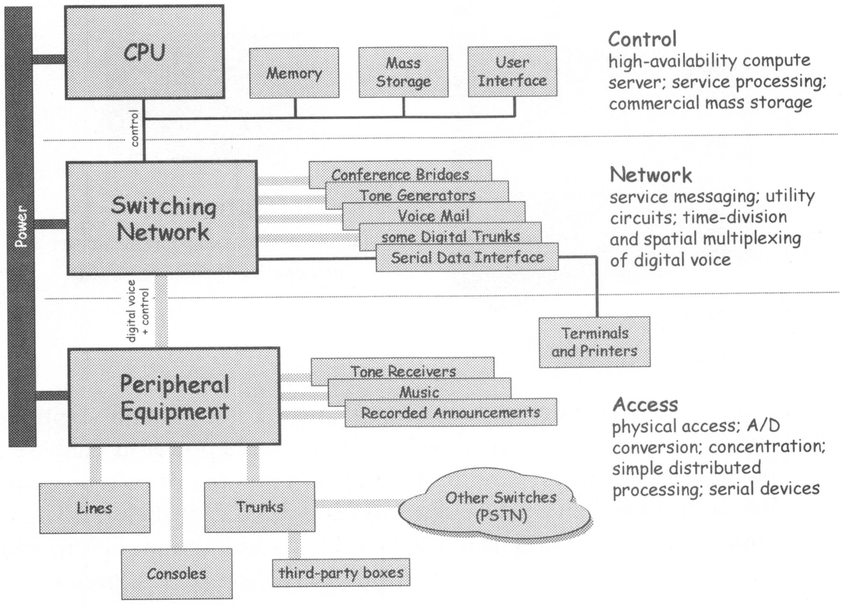
Today, the M1 comes in three basic shapes. Large systems (Options 51C, 61C, and 81C) are floor-mounted racks, with fans and power in the base and cabling out the back. The systems are built up as a number of columns, each of which has up to four Universal Equipment Module (UEM) shelves. The Option 11C, targeted at smaller sites, is normally sold as a wall-mount unit to save the customer some real estate. The Option 11C Compact is (predictably) a scaled-down, pre-configured 11C system in an even more compact box.

A matrix detailing the sizes of our various current offerings can be found at http://www.nortel.com/entprods/meridian/products/options/Tech_specs.html.

1.5.2 Logical decomposition

Each generation of hardware re-thinks the physical packaging to keep up with high-tech fashion trends. However, taking a short step back will reveal that ever since the first SL-1, our PBXs have had the same conceptual layers: for cost reasons, many terminals are analog; terminal wires connect to peripheral equipment which converts signals from analog to digital, do some concentration, and may have some basic signaling ability; peripherals connect to network equipment, which switches these signals digitally; and the network equipment connects to a computing server, which controls the overall service. This model is fairly cost-effective, flexible, and powerful, and is nearly universal as the generic stored-program-control telecom switch design.

We can evolve each of these main units more or less independently, and customers can purchase different levels of capability for each, to trade off grade of service against cost. The more powerful the CPU, the more calls and services it can handle. The bigger the network, the more simultaneous calls can be up. Fancier peripherals can support fancier terminals, and may offload the CPU.



Note that although a small system may combine all of these functions into a single cabinet, the *logical* decomposition is still essentially the same.

1.5.2.1 Power

The power subsystem is very dependent on the configuration. It may take 110V AC, 220V AC, or -48V input, depending on the environment. Redundant configurations usually have substantial battery back-up.

1.5.2.2 Control

Call processor (CP)

This is the heart (well, brain) of the traditional "Common Equipment (CE)", where all of the high-level call processing, maintenance, and user interface functions are performed. On older systems and Option 11Cs, the CP even does ringing and tone timing. Until the "X-Calibur" project, this was the only real computing engine in the entire system, so it was hardwired into the Network Equipment (NE), and controlled NE behavior through a group of hardware registers. Successive generations tended to move more computing power and messaging bandwidth lower into this structure, allowing service functions to migrate this way as well.

"Redundant Configurations" (Options 61C and 81C, and the older 61, 71, and 81) have dual CPs. The inactive CP waits in warm-standby mode, in case of failures of the primary CP.

Memory

Each configuration has a slightly different combination of on-chip cache, L2 cache, SRAM, DRAM, shadowed DRAM, and Flash ROM, based on the technology available, the speed required, and the amount of redundancy expected.

Storage

Each machine has some subset of: PCMCIA Flash ROM, tapes, floppies, (possibly shadowed) hard disks, CD-ROMs, or even off-board storage via an Ethernet port. The choices are based on the available technology, size of loads, reliability, speed, and system cost.

User interface

In earlier systems all Operations, Administration, and Maintenance (OA&M) was controlled by teletype-style terminals connected directly to the Common Equipment via a serial port. Later generations often used more sophisticated management tools and Application Processors connected to an Ethernet LAN port on the Common Equipment shelf, or had Serial Data Interface cards on the Network Equipment.

1.5.2.3 Network

The Network layer connects streams of digital data, and routes control messages around the system. On the Meridian 1, the Network Equipment (NE) performs these tasks.

The network "loops" that carry speech between NE and Peripheral Equipment (PE) have changed over the years, but generally contain 32 channels of 64-kilobit PCM voice each. The usual arrangement is to multiplex the channels at the bit level (one bit per time-slot per cycle). Mixed in with the payload data, signaling data is carried between terminals, and between the Core and the PE.

M1 NE tends to be engineered to have a fairly high blocking ratio; a loaded switch has about four times as many terminals as network connections. This is fine as long as many terminals are idle at any given time, but still requires a bit of planning to get the load balancing right.

A fully-populated network module, connecting 16 loops of 32 channels each, is known as a half group. A pair of these network modules comprises a full group.

Besides the expected switching function, many sorts of digital trunks and service circuits (like conference bridges and tone generators) are also hosted by the NE. The obvious reason for putting service circuits on the NE shelf instead of the PE shelf is because it allows fast non-blocking connection to the switching fabric. The more subtle reason is that historically there has been much better computational power available on the NE than on PE.

Original network equipment

Each network card can switch one 32 × 64-kbps channel network loop. They can co-exist with E-Nets.

Enhanced Networks (E-Net)

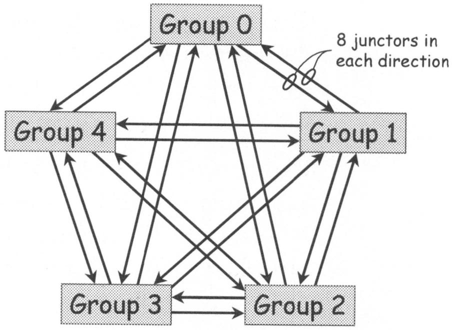
Each pack terminates 2 network loops. The time switching now uses random timeslots, rather than being based on fixed matched pairs.

Extended Networks (X-Net)

Each pack terminates a "Superloop" equivalent to 4 original network loops, and now has some intelligence in the peripheral shelf. X-Nets only talk to Intelligent Peripheral Eqiupment (IPE), discussed below.

Inter-Group Switch (IGS)

On our largest machines (Options 71, 81, and 81C), several network groups are connected to each other with dedicated junctors. Each junctor extends 1 network link (32 timeslots) in a dedicated one-way path from one group to another. The IGS is a space switch, composed of 8 junctors in each direction between each pair of network groups in the system. It can currently connect up to 5 network groups, although this is likely to increase to at least 8 in the near future.



1.5.2.4 Small System Controller (SSC)

The SSC, also known as the SCORE (System Core) card, is an all-singing, all-dancing processor board for the Option 11C that combines call processing control, non-blocking network switching, and both Ethernet and serial access functions.

The board currently has an MC68040 CPU, with a companion MC68360 for I/O processing and timing, an auxiliary MC68020 for tone generation and security, a DSP for tone detection, and an A31 IVD Message Handler to signal to its Peripheral Equipment. The SSC uses PCMCIA flash ROM cards instead of disks, and has a daughtercard architecture for additional memory or fiber connections. Up to two extra cabinets can be connected using plastic fiber (within 10 meters) or glass fiber (up to 3 km away).

The SSC runs most of the same software as our larger systems do, but with certain machine-specific changes `IFDEF`'d into it. The main Call Processor does a bit more work than a large system's CP (eg: tone timing) but for fewer agents, so it works out okay. By reusing the same software, Option 11C customers get all of the same features as larger systems, thus reducing our development and documentation costs and providing a consistent look and feel for customers who upgrade or have multiple systems.

Complete hardware details are at http://47.147.74.2/BVWTraining/docs/11C_hardware_slides.doc

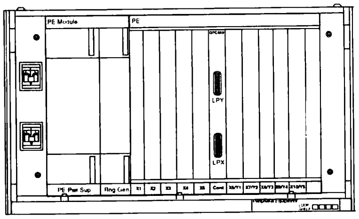
1.5.2.5 Access shelves

This layer provides distribution, concentration and media adaptation. All of the diverse access types for payload data streams discussed in this chapter reach the switching function provided by Network Equipment by way of an access shelf. On small systems, these cards plug into the main cabinet.

Earlier generations of M1 had almost no intelligence available to the Peripheral Equipment. Furthermore, there was a degree of concentration done between the NE and the PE. Therefore, digital trunks and some service circuits were placed directly into the NE shelves.

Enhanced Existing Peripheral Equipment (EEPE)

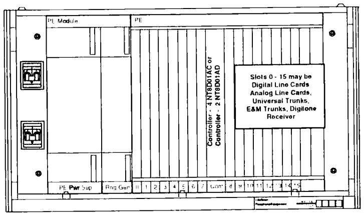
These are quite old now, but are still deployed because IPEs can not support certain old terminal types. The key line card is the:



- **SL1LC**: supports "SL-1" set line card + old attendant consoles

Intelligent Peripheral Equipment (IPE)

Most of the X-Calibur cards (X-...) live on IPE. Typical IPE cards include:


PE Module
PE
Controller - 4 NT8D01AC or
Controller - 2 NT8D01AD
Slots 0 - 15 may be Digital Line Cards Analog Line Cards, Universal Trunks, E&M Trunks, Digitone Receiver
PE Pwr Sup
Rng Gen
Cont

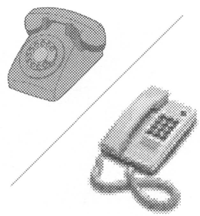
- **XPEC**: Peripheral Controller – receives signaling and control information from the network shelf over the DS-30X links, and returns status information.
- **XALC**: Analog Line Card
- **XDLC**: Digital Line Card
- **XMLC**: Extended Message Line Card, supports Message Waiting lamp
- **XEM**: Ear & Mouth trunk Card
- **XUT**: Universal Trunk Card, for Central Office connections
- **XDTR**: Digitone Receiver (detects in-band signals)
- **MGate**: DS-30X access, especially for voicemail, IVR, etc.

1.5.2.6 Line access: Terminals

The following is a sample of the phones we sell with the Meridian 1. Many of these are also sold with other Nortel products, notably the DMS family of central office switches. Also, other companies produce clones of many of these sets that will work with M1.

Analog sets

These are your Plain Ordinary Telephone Service (POTS) phones, although they sometimes support the Peculiar and Novel Services (PANS) like Call Waiting, developed in later years. They have both analog voice and analog signaling. The "500" set has a rotary dial (your grandmother's phone), while the "2500" set has buttons, but other than that they are basically identical. They are known in the code as PBX sets. The M1 can also host "CLASS" sets (really designed for home use) but we would normally prefer to sell people digital sets.



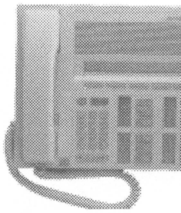
SL-1 sets

Also known as a Business Communications System (BCS) set, this is the original business set first shipped in 1975. It looks a bit clunky now, but it's still reasonably useful. It uses six wires to provide analog voice and (slightly sluggish) digital signaling. SL-1 sets have been superseded by digital sets, and we no longer sell any equivalent.

Digital sets

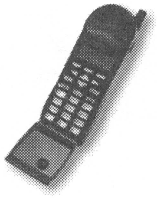
The "Delta II" and "Aries" families of sets, also known as Meridian Modular Terminals (MMT), have both digital voice and digital signaling. Some of the variables are display size, display language, number of buttons, soft keys, headset connections, and external data ports. "Real soon now", the M9617 set will come with a USB port and a compact disc full of PC software to drive CTI applications.

Basic Rate Interface (BRI) sets



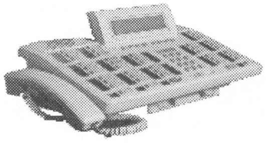
Basic Rate Interface is the station-style version of Integrated Services Digital Network (ISDN) access. A BRI set terminates "2B+D"—two 64-kilobit "Bearer" channels, theoretically for voice + data, and one 16-kilobit "Data" channel for signaling. It's a nice option for work-at-home, although there are some better ones under development (eg: cable modems, xDSL, etc.) and it's fairly unconvincing as an alternative to an office LAN.

Mobile sets



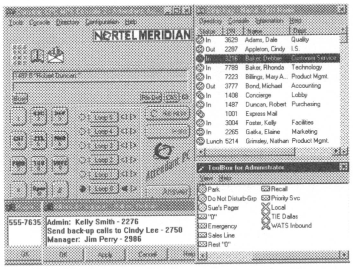
There are a number of options for mobile handsets. At the low end, there are Companion and Meridian series Cordless Telephone (CT) sets, which allow you to move a few hundred yards from your desk. At the higher end, Companion supports several cellular standards, for which users can choose terminals from Nortel or other vendors.

Attendant consoles



There are a number of variations on the original attendant console, but they all do essentially the same jobs, and most of these jobs are heading towards obsolescence. About the only thing attendants still do much better than electronics are ever likely to is provide a warm human response to an external caller, and this is sufficient to ensure their survival for at least a little longer.

PC-based consoles



Status	DN	Name	Dept
In	3629	Adams, Dale	Quality
Out	2287	Appleton, Cindy	I.S.
In	3216	Baker, Debbie	Customer Service
In	7789	Baker, Rhonda	Technology
In	7223	Billings, Mary A...	Product Mgmt.
Out	3777	Bond, Michael	Accounting
In	1408	Concierge	Lobby
In	1487	Duncan, Robert	Purchasing
	1001	Express Mail	
In	3004	Foster, Kelly	Facilities
In	2265	Gatka, Elaine	Marketing
Lunch	5214	Grimsley, Nathan	Product Mgmt.

555-7635

Admin: Kelly Smith - 2276
Send back-up calls to Cindy Lee - 2750
Manager: Jim Perry - 2986

Park
Do Not Disturb-Grp
Sue's Pager
"0"
Emergency
Sales Line
Rest "0"
Recall
Priority Svc
Local
TIE Dallas
WATS Inbound

Take the M2250 console shown above, remove the user interface, re-box it to sit under a PC monitor, and add some Windows™ message center software to drive it, and you'll have the **Meridian 1 Attendant PC**. In addition to all of the usual console features, it includes dial-by-name and drag-and-drop speed call.

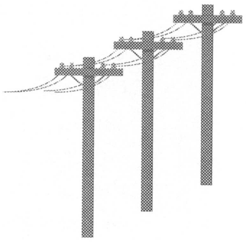
CPLUS[4] was an earlier, DOS-based equivalent.

[4] No relation to C++.

Data terminals

The **Meridian Communications Adapter** (MCA) allows data devices to communicate through the telephone network. Personal computers, ASCII terminals, video equipment and Group IV facsimiles can connect to the M1 via the built-in RS-232 or V.35 interfaces by adding an MCA to a Meridian Digital Telephone. This product includes the 19.2 K and 28.8 K asynchronous MCAs, and the 64K synchronous MCA. Also, an **Analog Terminal Adapter** (ATA) has just been released, allowing users to attach fax machines, computer modems, or 500/2500 sets to a digital set, and use both *simultaneously*.

1.5.2.7 PSTN access: Trunks



Trunks are our links outside of the M1; they provide speech and signaling paths to other switches around the world. The Meridian 1 supports a plethora of trunk types. The details depend on what country you're in, how much money you want to spend, and how many calls you need to support.

Historically (right back to Alexander Graham Bell) trunks were analog; they were basically an extra-long pair of speaker wires. To this day, many trunks still are analog, and we support several variants. Analog trunk standards differ by the way they use such things as in-band tones and electrical polarity to achieve signaling; by the services they support using those signals; and by whether they use two wires (carrying speech in both directions at once, with a bit of echo) or four (each pair carrying speech in one direction, much better for long distances).

Digital trunks make it easier to multiplex many conversations at a time onto a single pair of wires, will span arbitrary distances without degrading speech quality, support more interesting signaling, and can haul data and video as easily as voice, so they are by far the most common for modern long-distance trunks. All digital voice trunking (and for that matter digital switching too) starts by converting the analog audio signal to a digital data stream using Pulse-Code Modulation (PCM)[5].

[5] Math weenie details on PCM: The basic technique is *non-uniform logarithmic quantizing* of the waveform: quantizing to convert a continuous amplitude signal to digital; logarithmic (like decibels are) because it's a good match for how our ears hear volume; but non-uniform because $log(0)=-\infty$, which makes silence hard to represent. The human ear can hear frequencies approaching 20,000 Hz, but normal speech doesn't include much higher than 4,000 Hz. Nyquist's Theorem says you have to sample at twice the frequency of the waveform you're trying to catch, so we sample at 8,000 Hz (although your CD player needs a much higher sample rate to get the piccolos). Bytes are a convenient size to make each sample, and turn out to be good enough, so the total bandwidth needed for a one-way speech path with no extra compression is 64 kbps.

A-Law is the ITU-T's standard for PCM encryption; μ-Law (pronounced "myoo law") is the North American equivalent. We can support either.

From an M1's point of view, digital trunks come in two speeds: 1.5 Mbps (24 × 64-kbps channels), known as T1, which is the standard in North America and Japan; and 2 Mbps (32 × 64-kbps channels), known as E1, which is the standard everywhere else. Higher speed connections, like T3 (45 Mbps) and OC-3 (155 Mbps), are now the norm in the public network. We probably ought to start supporting them now for big Option 81Cs and busy call centers, and we will certainly need them if we ever increase the bandwidth we deliver to the desktop. Unfortunately, doing so well would require a lot of other rework, most notably a high-speed access shelf of some description.

The service protocols which run on top of digital trunks can be grouped into Channel-Associated Signaling (CAS) and Common Channel Signaling (CCS). Put simply, CAS has a dedicated signaling subchannel for each voice path on the wire, either in a dedicated signaling channel or mixed into the voice stream. Most of the CAS trunks we support are national variants of "R2" signaling. CCS puts all of the signaling for a carrier into a separate common stream which may be routed differently from the voice payload, and supports a number of cost-reducing or even revenue-generating features. Within M1, we usually [mis]use the term "ISDN" to refer to all CCS trunks. We support DPNSS, DASS2, QSIG, and MCDN for private networks; and ETSI or ANSI PRI, and dozens of national CCS7 variants for public networks. Most of these come in both T1 and E1 sizes.

In all, we support the standard trunk interfaces of over 100 countries, which at the moment is one of our key competitive strengths.

At the receiving end of a trunk, poor impedance matches can cause a portion of the transmitted signal to be reflected back towards the source. On long-haul connections, you can hear this as an echo, and it can become surprisingly difficult to talk or even think clearly in its presence. Analog trunks use echo suppressors, which try to determine which end of the trunk is speaking and insert loss in the opposite direction. Digital trunks use echo cancellers, which digitally subtract a predicted echo signal from the received stream. Both techniques improve voice quality, but may interfere with fax and data calls. Therefore, since the M1 can't (yet) automatically route data calls differently from voice calls, customers usually have dialplan options that let them access trunks with no echo treatment (eg: our 6-500-ESN dialplan[6]).

[6] See http://47.141.5.216/ESN/documentation/plan.html for more details.

The physical carrier for a trunk signal may be copper, infrared transmission, fiber-optic cables, microwave radio, or satellite links. Each has different engineering pros and cons, depending on the physical, economic, and regulatory environment of the customer, but each terminates on the M1 as some sort of trunk card.

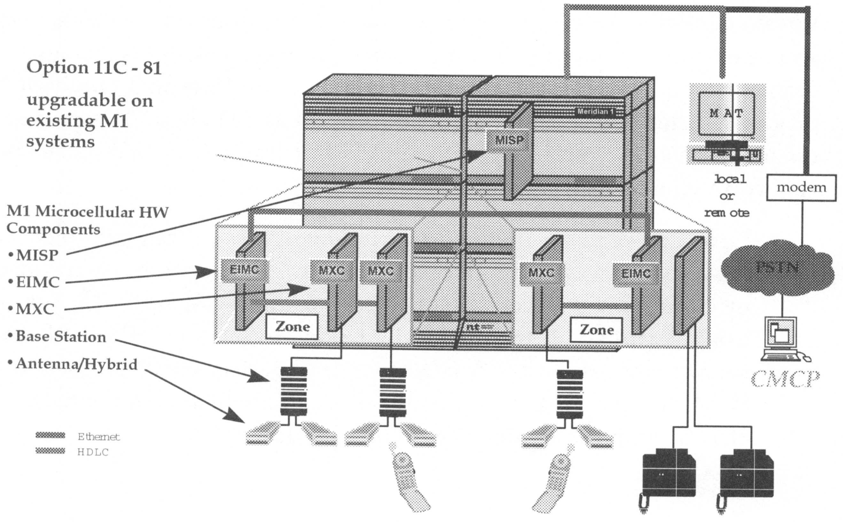
Finally, the following section will discuss several types of "service trunks", which provide facilities like loudspeaker paging and recorded announcements. These are not true trunks, in the sense that they do not provide external connections to other switches, but they tend to interact with call processing hardware and software in the same manner as normal trunks would.

1.5.2.8 Wireless access: Companion

The Companion brand name really covers a collection of technologies that support various cordless terminals with the M1 in various markets around the world.

The **Companion Microcellular** service allows standard digital cellular telephones to be used for in-building wireless operations. This has a number of advantages compared to public cellular service. There is no charge for air time, the phones operate at lower power, and many more phones can be operate simultaneously in the same physical area, without requiring people to carry two handsets. But to make this work bandwidth must be licensed from the local cellular companies.

The microcellular service is composed of a collection of components:



Companion Meridian Digital Enhanced Cordless Telecommunications (**MDECT**) is a similar in-building service made to the European DECT standard. It uses different but similar hardware and software, and has essentially similar user values: roaming, hand-off, decent voice quality, and some security. Companion C200 "Unlicensed" Personal Communications Systems (**UPCS**) is yet another flavor, based on yet another set of standards for the North American market. Unlike the Meridian Microcellular controller, the C200 works with more or less any switch. Again, the user values are similar, and C200 handsets are lighter, but they can't be used with the regular cellular network when you leave the building

The inner details of the wireless access hardware are not too important for most designers. However, you should remember that a user's terminal is *not* permanently mapped to a given voice port (because it's mobile! ☺) and ends up routing through a Multi-purpose ISDN Signaling Processor (**MISP**) in a way that resembles internet access; but that a lot of the regular digital set software is reused to provide most services; and that the substantial extra processing power is provided by extra controller packs, eg: the Embedded Intelligent Mobility Controller (**EIMC**) and Microcellular Transcoder Card (**MXC**).

Finally, recent enhancements to Companion have started to deliver Computer-Telephony Integration (CTI) features to wireless users. Although the current user interface is inherently slim, this may prove to be a powerful combination in some environments like factories and hospitals.

1.5.2.9 WAN access: Bit conservation

Within a building or campus, the bandwidth over your own wires is very cheap, so there is little to gain from compression technologies. This changes dramatically as soon as you leave the campus. The **Passport** family of Nortel products includes a version packaged to fit on an M1 shelf. Passport connects an organization's LAN and PBX into an ATM or frame relay WAN backbone, to give you two major advantages:

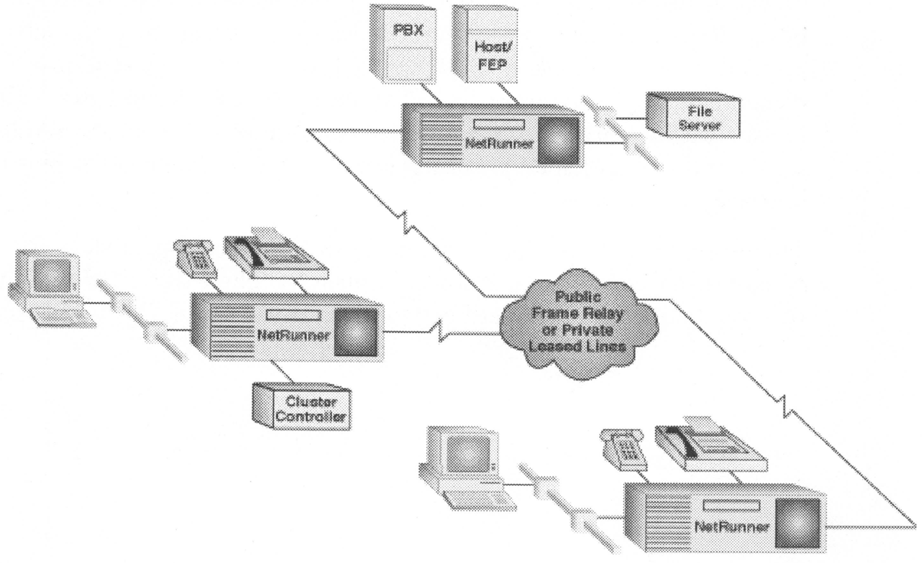
- Lower voice networking costs: Meridian Passport can cut voice trunking costs in two ways. It reduces the bandwidth needed to carry voice traffic by compressing a voice channel from 64 kbps down to 32k, 24k or 16k using Adaptive Differential Pulse Code Modulation (ADPCM) and silence suppression; and it consolidates voice with data traffic into a common WAN using dynamic bandwidth allocation, which is much more cost-effective than using dedicated voice links.

Multimedia wide area networking: By adding Meridian Passport to your Meridian 1 systems, you can build a WAN to handle voice, data, image and video traffic. You can also use Meridian Passport to consolidate all multimedia traffic generated at a Meridian 1 location and deliver it to a larger existing WAN.

This product pits two Nortel lines of business against each other (Broadband vs. Enterprise Networks) to earn revenue from a given customer, and so presents special challenges in terms of actual sales, but the technology seems solid enough.

The **NetRunner** and **Marathon** voice-over-IP gateway families, made by MICOM (which we now own) are an alternative to Passport. They cut costs by consolidating and compressing voice, fax, legacy data, and LAN traffic before sending it out as TCP/IP over public n × 64-kbps or T1/E1 Frame Relay circuits. They are low-end alternatives to Meridian Passport, and currently lead the industry in sales in their market segment. MICOM also has a number of other voice compression technologies that we are likely to leverage over time.

An enterprise network using NetRunner might look something like this:



1.5.2.10 Control access: Signaling links

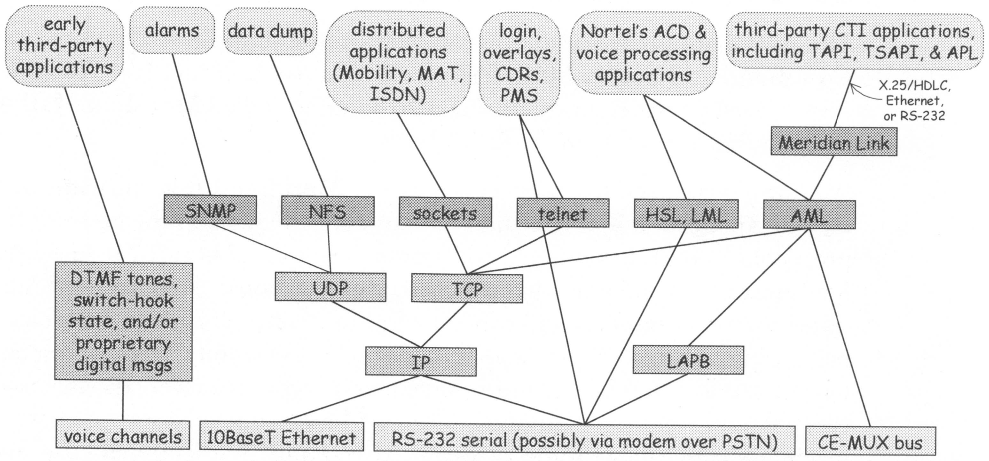
The payloads normally carried by PBXs are streams of 64-kbps digital voice data, sometimes known as "bearer" or "traffic" channels. These streams enter the M1 through terminals attached to Peripheral Equipment, are switched to other terminals by Network Equipment, and are never seen by the Call Processor. However, for the CP make the right decision about which pipes the NE should connect, it must receive signals from the outside world. In the traditional case, this is done by a phone going off-hook and dialing digits, which were carried in-band in the payload voice stream.

However, there are a number of off-board boxes which would like to have more sophisticated discussions with the M1 call processor. The boxes themselves will be discussed in Section 1.5.3, but first I'll give a brief overview of the connectivity options. (There's a chicken-and-egg problem with this discussion; please bear with me.)

The earliest off-board call manipulation technique, and one that is still used, especially by third-party vendors, is telephone emulation on traditional voice ports. Rather than having a special, dedicated signaling channel, each port pretends to go off-hook, dial DTMF digits, or maybe even send proprietary digital messages. This approach is automatically scalable, and gives full access to the standard feature set of the terminal being emulated (typically a M2616 digital set). On the other hand, it is nearly impossible to define more sophisticated messages and services.

Therefore, the M1 also supports a range of higher-level protocols, for both system management and call control signaling. These in turn are stacked on top of several underlying standard network protocols:

M1 ↔ auxiliary processor signaling links


early third-party applications
alarms
data dump
distributed applications (Mobility, MAT, ISDN)
login, overlays, CDRs, PMS
Nortel's ACD & voice processing applications
third-party CTI applications, including TAPI, TSAPI, & APL
X.25/HDLC, Ethernet, or RS-232
Meridian Link
SNMP
NFS
sockets
telnet
HSL, LML
AML
DTMF tones, switch-hook state, and/or proprietary digital msgs
UDP
TCP
IP
LAPB
voice channels
10BaseT Ethernet
RS-232 serial (possibly via modem over PSTN)
CE-MUX bus

The physical connection to the M1 depends on the speed required and the vintage of the equipment at each end. 19.2 kbps serial connections used to be "fast". Today, 10BaseT Ethernet or OC-3 fiber would probably be the medium of choice. Small systems may opt to use the shelf's inter-card backplane (the CE-MUX bus) directly, to save additional cabling.

Early applications assumed that they had a dedicated point-to-point cable, and tended to build their own messages directly on top of the physical medium. Examples of this include traffic monitoring, Call Detail Recording (CDR) for billing, Property Management Systems (PMS) for hotels, and the original TTY user interfaces. The **High Speed Link** (HSL), now a misnomer since access is only at 19.2 kbps, is used by Meridian Max to track and direct the progress of calls through a group of ACD queues. A typical ACD call would have five or more HSL messages. Max then uses the **Load Management Link** (LML), also known as the Configuration Control Link (CCL), to manage performance by changing the datafill which controls how calls get routed.

Application Modules (described in the Section 1.5.3.10) communicate with the M1 core equipment over our proprietary **Application Module Link** (AML). AML is a call control protocol designed to manage calls going to a group of similar agents (like operators in an ACD group or ports on a voicemail system). It is usually carried using Link Access Procedure-Balanced (**LAPB**) over a serial port on the older Enhanced Serial Data Interface (ESDI) or Multi-purpose Serial Data Link (MSDL) card, or more recently on TCP/IP over Ethernet. For IPE Modules, the AML messages are simply carried on the CE-MUX bus. AML drives services provided by CCR, M911, ICCM, Meridian Mail, Meridian Link, and VISIT, all described in the next section.

The AML hardware also supports an older subset of the AML protocol called **Command and Status Link** (CSL) for talking to Meridian Mail. And the MMail system itself provides an external control interface called **Meridian Mail Access** using RS232 Service Module (RSM) cards.

AML use is reserved for Nortel equipment. **Meridian Link**, the name of one type of Application Module, and (perhaps confusingly) also the name of the protocol it speaks, lets other "host" computers control both AML and Meridian Mail Access to build integrated voice/data applications. Meridian Link lets the other computer monitor and control telephone sets, call routing and voice processing features. Typical applications include customer service centers, dealer locator services, hot lines, fund-raising, market research, reservations, brokerage services, and emergency services. Meridian Link is carried over the host machine's native data protocol: X.25/LAPB (HDLC) or Ethernet. We currently support more host computer middleware packages than any other PBX in the industry, including:

- Dialogic's CT Connect
- Digital Equipment Corporations CIT
- Genesys's Labs T-Server
- Hewlett-Packard (HP) Applied Computerized Telephony (ACT)
- International Business Machines (IBM) Call Path
- Microsoft Windows NT Telephony Application Programming Interface (TAPI) 2.0
- Tandem Call Applications Manager (CAM)
- Nihon Unisys

MMCS also uses an Ethernet Meridian Link connection to control its AIN-style call routing, rechristening the service **Application Processor Link** (APL).

We also support Novell's Netware Telephony Services API (**TSAPI**) standard, but the configuration is different, and is based on the Meridian Communications Adapter (MCA).

These days, the physical carriers tend to be shared by a suite of higher-level protocols, each tailored to a different job. The user interface might log in via telnet. The System Management Platform (SMP) uses telnet too, but can also perform request/response style transactions using VxWorks or RogueWave **sockets** (`Net.h++`) over Transmission Control Protocol (**TCP**). Database archives and software delivery use Network File System (**NFS**) over User

Datagram Protocol (**UDP**). Alarms are signaled to the Event Monitor application using the industry-standard Simple Network Management Protocol (**SNMP**) protocol over UDP. We also use SNMP for the session management code that controls logon.

1.5.3 Bells and whistles

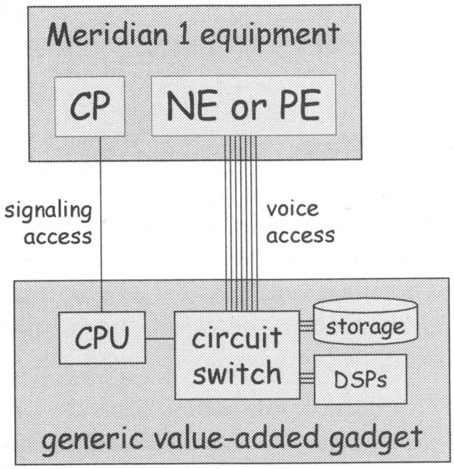
The following devices are roughly the latest of several generations of products which have performed services for the M1. Each will doubtless be replaced by newer, faster, cheaper, smaller, niftier versions as time goes by, but the important thing is to at least be aware that the services exist, and that they are sort of separate from the heart of the switch. Each performs a generic service that is reused in multiple applications.

Many of the products in this section have versions that are "in-skins": that is, they are so closely linked to our architecture that they can be placed inside the standard M1 cabinetry, and our customers may not think of them as separate units at all. Others are built from hardware and software that sits beside the M1, and uses the access and signaling pipes just described to send and receive data streams. Some may even be located remotely.

Because these systems are a bit removed from the M1, they are more likely to continue to work as customers upgrade their M1 software and hardware, or even alongside our competitors' switches. The downside is that for exactly the same reasons, they have more direct competition from third-party equipment manufacturers. For instance, many M1 installations have voicemail systems that were not built by Nortel.

If our future market became dominated by the "unPBX", then these extra gadgets probably constitute many of the important ways that a "call server" (analagous to today's file server or printer server) could add value to the voice switching function that TCP/IP over Ethernet would steal from the M1 switching fabric. The other key piece a good unPBX needs would be a PSTN gateway so that people not on the internet could talk to you. This might conceivably look a lot like today's Peripheral Equipment.

You'll notice that almost all of these gadgets are variations of the following simple model (which itself is only a slight variation on a PBX):



We might keep on the lookout for opportunities to merge some of them, although we need to balance the advantages of fewer platforms against those of cheap, plug-and-play style, single-function boxes.

1.5.3.1 Conference bridges

Mixing together PCM voice requires a bit of careful digital signal processing, and rather than trying to do it in the network, we connect each party in a conference call to a port on some sort of conferencing circuit, the latest vintage of which is the **Meridian Integrated Conference Bridge** (MICB). The MICB is a single-card, 32-port bridge that can handle up to 10 simultaneous conference calls. It fits into a single IPE slot and merges the digital voice streams together magically so that everybody hears the correct set of other speakers. It also has built in tones and multi-lingual voice prompting.

Customers buy the MICB in one of four capacity options (12, 16, 24 and 32 ports), but this is just a software-controlled behavior, and additional ports may be activated by software "keycodes". If more than 32 ports are required, multiple cards may be used. Having a single hardware form makes our ordering and manufacturing processes simpler (and therefore cheaper).

This card is used to support a range of conference styles, including 3-way call, meet-me conference, chairperson-controlled conference, and attendant-controlled conference.

1.5.3.2 Recorded announcements

Recorded announcement (RAN) machines are used to play back voice messages to both internal people (say, to tell them they've dialed a wrong number), and to external callers (for things like your business hours and locations). **Meridian Integrated Recorded Announcements** (MIRAN) can support up to 10 messages, and 40 simultaneous playback channels. Like the MICB, we sell it in a range of capacity options (20, 36, or 40 concurrent calls), but the hardware is the same for each, and the upgrade is just removing the hobble with a software keycode. For even greater capacities, an M1 can support up to 16 MIRAN cards, but a better solution is to use the card in the new broadcast mode, whereby many listeners can be connected using one-way speech paths to a single MIRAN port, allowing a single card to support 300 listeners.

By default, a MIRAN can record only four minutes of speech, but this can be upgraded to over 5 hours using plug-in PCMCIA cards. Special features include time-of-day messaging, remote recording, and simplified OA&M. The card uses some cool newfangled dual-sided surface mount technology.

As with many of the other "bells & whistles", some customers use external, third-party boxes, or older generations of Nortel equipment, to do this job.

1.5.3.3 Music boxes

Music "trunks" are close cousins of recorded announcement machines. Like RANs, they get used when you don't yet have a better place to send a caller. Unlike RANs, you can send a caller to music at any time, rather than having to align with the start of a message. There are many variations, which may end up using radio, tapes, flash ROM, or even a simple synthesizer as the music source. The software adds music to a call much like it would make any other trunk connection.

1.5.3.4 Voicemail

Meridian Mail is a family of voicemail products (now "*Multimedia Mail*", which seems to mean voice, fax and text storage), ranging from a single-card option (12 ports, 48 hours of voice storage) optimized for Option 11C, to the monster free-standing Message Services Module, with up to 192 ports, 42,000 mailboxes, and 2400 hours of shadowed voice storage. The whole family tries to have an identical end-user interface. It was the first major addition to the original switch, and is also able to work with DMS-100 and non-Nortel products, although certain features work best on the M1. In the near future, **Meridian Communications Exchange** (MCE) will replace it.

Meridian Text Telephony System (MTTS) uses some of the multimediality of MMail to help deaf people communicate. If a deaf person calls in using a TTY/TDD terminal, MMail can originate a voice call to the required person, play them a message indicating that a TTY/TDD call is waiting so that a chat session can begin, or take a message if the person is unavailable.

Visual Interactive Technology (VISIT) Messenger is a client/server product that allows users to manage voicemail, email, and faxes with a single email-style indexing system. Regardless of medium, you can see who sent the message, when it was sent, and its status. Because its desktop client can take full advantage of desktop GUIs, Meridian Messenger allows much more efficient access to standard voicemail features. It also allows store-and-forward manipulation of faxes, as well as allowing email to be sent out as fax to people without email facilities. **VISIT Assistant** is sort of call center for one: call queuing, prioritization, and redirection for a single telephone, driven by a desktop GUI.

VISIT Messenger, VISIT Assistant, Meridian Integrated Voice Resonse (IVR), and third-party voice mail service providers can control the Meridian Mail system using the **Meridian Mail Access** product. Connection to the Meridian Mail system is provided using a serial connection to either the Meridian Mail RS-232 Service Module (RSM) card or through the serial port of one of the CPU modules.

Meridian Mail Net Gateway (MMNG) connects MMail to a TCP/IP WAN using the MVIP protocol. This is such an obviously good way to send voice over IP that it's amazing it wasn't done sooner. The Internet may be a fairly lousy way to send real-time data, but it's excellent for the kind of asynchronous transfers that voicemail systems do. When MVIP is used for controlling MMail, PC client applications like Meridian Messenger can access all of the MMail features, but using simple GUIs instead of user-hostile dialed-digit combinations.

Meridian Mail Reporter is a Windows application that improves the manageability of a Meridian Mail system. By gathering and sorting data, Meridian Mail Reporter can help make a Meridian Mail system run more efficiently, protect owners from toll fraud, and allow them to bill back special features such as fax on demand to specific departments.

1.5.3.5 Integrated Voice Response (IVR)

IVR offers an extensive range of advanced capabilities, including outdialing, Audiotex, fax-on-demand, fax broadcasting, interactive fax (for fax order confirmation, bank statements by fax, etc.), discrete and continuous speaker-independent speech recognition, text-to-Speech, Analogue Display Services Interface (ADSI), IBM and DEC host connectivity, and SQL database support.

IVR technology allows companies to employ fewer operators by automating repetitive or routine telecom activities. IVR systems are often used to front-end big call centers, so that real operators only talk to people who have already identified who they are and why they're calling. At it's best, IVR provides end-customers with quick access to information at any time of the day or night, from any push-button phone. You'll recognize the following typical IVR uses:

- Health care—physician referral, appointment scheduling, test result reporting, appointment confirmation
- Banking—account balance and interest rate inquiries, loan applications
- Catalog sales—order entry, order status, inventory inquiries
- Customer service—order status, service dispatch, product or repair information
- Human resources—scheduling, benefits inquiries, employment opportunities
- Education—class registration, special event ticket purchases, tuition payments
- Insurance—claims status, eligibility
- Reservations—airlines, hotels, travel agencies

Integrated IVR (alias Meridian IVR) includes four main components: a GUI-based application generator, a system administration pack that allows installation on a live switch, a report generator, and a voice prompt editor. It takes advantage of detailed knowledge about the M1 interfaces and calling features, but is therefore less flexible.

Open IVR provides flexibility for increased functionality and the capability of working with both Meridian and non-Meridian switches.

The **Integrated Meridian Speech Directory** (IMSD) allows a user to "dial" by speaking the desired party's name into the handset[7]. The initial applications focus on Corporate Directory Voice Dialing (CallAssist) and System Security (CallSecure). The project is known internally as Language Command Environment (LANCE).

[7] This service is sometimes known as "Ghostbusters". Try it at MPK by dialing 5555.

1.5.3.6 Dialing emergency services

"911" is the universal number for emergency services in North America (similar to "999" in the UK, or "112" in the rest of Europe). With the new legislation, PBXs are now going to be required to provide 911 capability and location-specific information regarding these emergency calls. Nortel is working with Telident, a third-party vendor in Minneapolis, to build the **Meridian 911** product that will address these changing needs. This product provides translation of non-DID numbers, comprehensive on-site notification, and database management and synchronization.

The portfolio includes the following three products:

- Station Translation System (STS)–hardware between the M1/MSL-100 and the outgoing trunks that carry the 911 calls
- TRAX On-site Notification–software application that resides in a PC attached to STS used to notify security staff about 911 calls in progress
- ShadowMAX Database Maintenance Software–resides in a non-dedicated PC and ensures that the data in the STS and TRAX is consistent and performs the necessary remote ALI database updating

1.5.3.7 Alarms

The system monitor card has several "dry contacts", relay inputs from external sensors (like sump pumps, intrusion detectors, and fire alarms) which allow the M1 to report environment problems, and relay outputs to control external devices, often connected to sirens or flashing lights to indicate serious outages.

1.5.3.8 Management tools

Meridian Administrative Tools (MAT) is the standard tool for system administration. It's a PC-based, GUI-style system that connects to the M1 over the Ethernet port. It provides Traffic Analysis, Call Accounting, Call Tracking, Alarm Management, Maintenance Windows, ESN Analysis and Reporting, and on-line NTPs. **Meridian Manager** was an earlier incarnation of this tool. As a consequence of these external boxes which rely on our user interface, we need to minimize changes to it.

Meridian 1 Network Management (MNM) for Spectrum (built by Cabletron) is one of a class of third-party systems that uses Simple Network Management Protocol (SNMP) over the Ethernet interface to do alarm management. It leverages the Release 22 "Open Alarms" package. We no longer sell MNMs, but there are a number of them in the field.

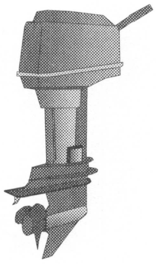
Several different **CDR post processors** are available from third-party vendors which take a stream of Call Detail Records and produce resource utilization or department billing reports. Every time we change the CDR format, we risk messing these up. Although we don't sell them, many of our customers rely heavily on them.

When a customer upgrades from one software release to the next, they normally use a separate PC-based application to do **database conversion** to correct the formats of any changed overlays. **Configurator** is a Windows NT application that helps advise on and coordinate database changes, and populates control tables on the M1. It also talks to Symposium TAPI server, and probably other boxes.

1.5.3.9 The other music

"Music" sort of stands for Multi-User System for Interactive Communications, but is also just a Rainbow-style code name for a soon-to-be-product. It combines a MUDdy web-browser interface, server-based persistency of calls and user profiles, and Symposium call processing to give a potentially-exciting new way of thinking about control of multimedia calls. We'll have to wait to see how much the market likes the product as shipped.

1.5.3.10 Off-board processing engines



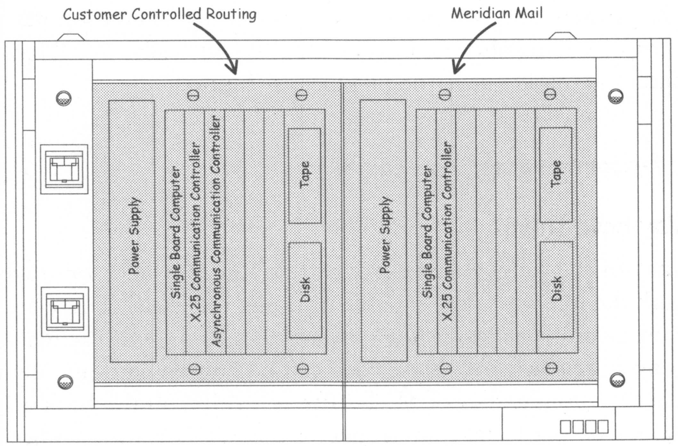
Why have them?

Many of the gadgets just described do things which you might think the M1 was quite capable of doing for itself. There are many advantages to putting new services outside of the main computing platform. New code can be written without delving into the complexities of SL-1 software, although the trade-off is that it's harder to take advantage of the M1's detailed service state knowledge. But precisely because of this the integration job is easier. Features can be sent out as soon as they're ready, rather than waiting for the whole M1 software release to be ready, but you have to build, ship and install new hardware for each site, instead of just installing some software. The main CPU remains free to work on delivering calls (which for an active call center or a big Option 81C is enough to keep it very busy), as long as the new messaging overhead is not to severe. Failure modes are more controllable, typically affecting only one processor. Customers may be able to leverage Moore's Law (which predicts the decline in cost of computing horsepower by 50% every 18 months), although the trade-off is that they have to buy an extra box in the first place. And tools like MAT and Meridian NAC may connect to multiple PBXs simultaneously.

Therefore, in addition to some highly-tailored cards and boxes that perform specialized roles, many of the functions just described are implemented on more general-purpose platforms. These platforms challenge our distribution ability (as Moore's Law makes obsolete machines we thought our customers would like), but the numerous advantages mean we'll stay in this business somehow or other for the foreseeable future.

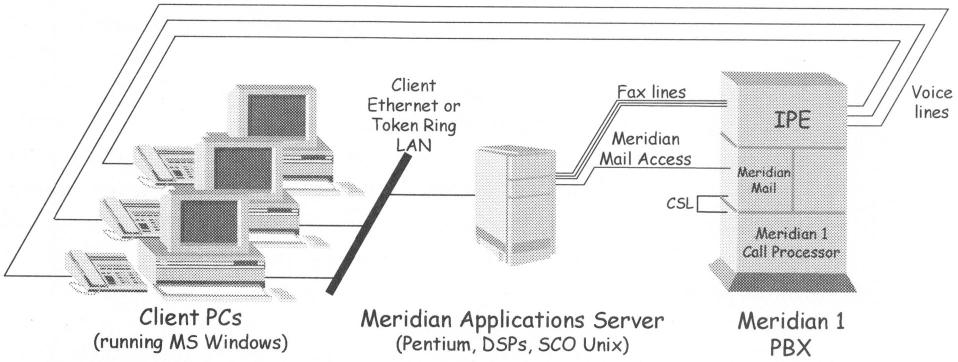
What are the platform options?

There's an inappropriately long list of platforms to choose from. Each combines a commercial CPU running a commercial operating system, some storage, usually a database, and assorted Digital Signal Processing (DSP) chips. We can rationalize the diversity by saying that they serve different niches, but thinking of ways to merge onto a smaller subset of platforms might be a better use of the mental energy.

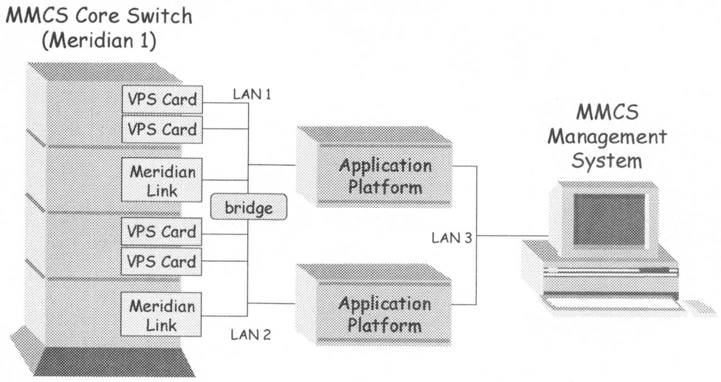

Customer Controlled Routing
Meridian Mail
Power Supply
Single Board Computer
X.25 Communication Controller
Asynchronous Communication Controller
Tape
Disk
Power Supply
Single Board Computer
X.25 Communication Controller
Tape
Disk

Two Application Modules in an AEM

Application Module (AM) is the name for the first family of boxes to provide generic auxiliary processing. They are built around an MC68030 or MC68040 single board computer sharing a VME bus with a disk/tape unit, a power supply, and a few extra cards. Depending on which additional cards are equipped, AMs provide communication paths to the M1 via LAPB/X.25, RS-232 serial, modem over 2-wire PSTN line for remote maintenance, Ethernet, or SCSI. If you put two AMs into a UEM shelf, you call it an Application Equipment Module (AEM). Alternatively, some AMs (called **IPE Modules**) have packaging that fits 4 slots in an IPE shelf and or 3 slots in an Option 11C cabinet. These tend to be about half the cost and lower capacity, and are well-suited to the small system market. Either way, the software running on them is the **AMBase** application sitting on a Unix platform. Among the applications available on an AM are CCR, MQA, MAX, NAC, Meridian Mail, Meridian IVR and Meridian Link. AMs came from the Mountainview lab (now MPK).


Client Ethernet or Token Ring LAN
Fax lines
Meridian Mail Access
CSL
IPE
Meridian Mail
Meridian 1 Call Processor
Voice lines
Client PCs (running MS Windows)
Meridian Applications Server (Pentium, DSPs, SCO Unix)
Meridian 1 PBX

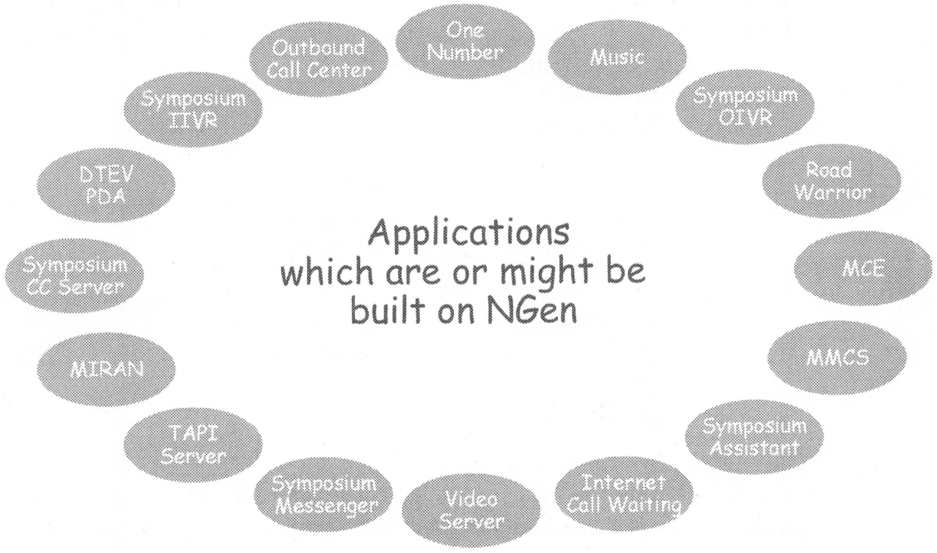
Meridian Applications Server (MAS), circa 1993, previously known as TeleWare, is a standard Pentium PC tower running SCO Unix. It connects to the M1 via an RS-232 port and analog lines (for fax and voicemail transmission), and to the customer's LAN using standard Ethernet or token ring. It is used to host VISIT Messenger and VISIT Assistant. The server connects to the Meridian Mail RSM card via short haul modems. MAS was built by the Minnetonka lab.



The MMCS **Application Platform** (AP), circa 1997, is a DEC "Prioris" rack-mounted Pentium board running SCO Unix with an Oracle database. APs handle most advanced services, like AIN call processing, flexible CDRs and SS7 signaling. The software platform on which their services are built is called APBase. The AP has no voice-handling ability, so applications like IVR are done by having the AP control additional MMCS **Voice Processing Systems** (VPS), yet another platform: a single-slot IPE card, industry-standard CPU, PCMCIA flash ROMs, various DSPs, 24-port M2616 emulation to M1. The AP and VPS were designed by Marne-la-Vallée.

The grand unifying theory of platform evolution is that one day soon, everybody will be on the **Next Generation Server** (NGen). This is the latest multimedia applications server family for the Enterprise Networks product line. It is based on a choice of Intel CPUs running Windows NT, with built-in DSPs and Multimedia storage. It ships either as a Tower Rack Platform (TRP), or as an IPE card. The idea is that it will provide a common scalable platform for the numerous Meridian applications that have previously been deployed on incompatible platforms. It speaks TAPI and ECTF (Enterprise Computer Telephony Forum), and PCI and PC Card (PCMCIA) cards can be added as necessary. The strategy is to buy generic equipment (CPU, O/S, bus, and some middleware) and build only specialized, high-value components where we think we have some competitive strengths (DSPs, applications). The problem is, even if it became true that NGen had the right price, capabilities, and robustness to serve all of the markets for which it claims to be useful, there would still be a huge installed base of older equipment that we would have to interact with, and that might even want feature upgrades. NGen is from Toronto, circa 1997.

NGen is or hopes to be a host platform option for a whole bunch of stuff:



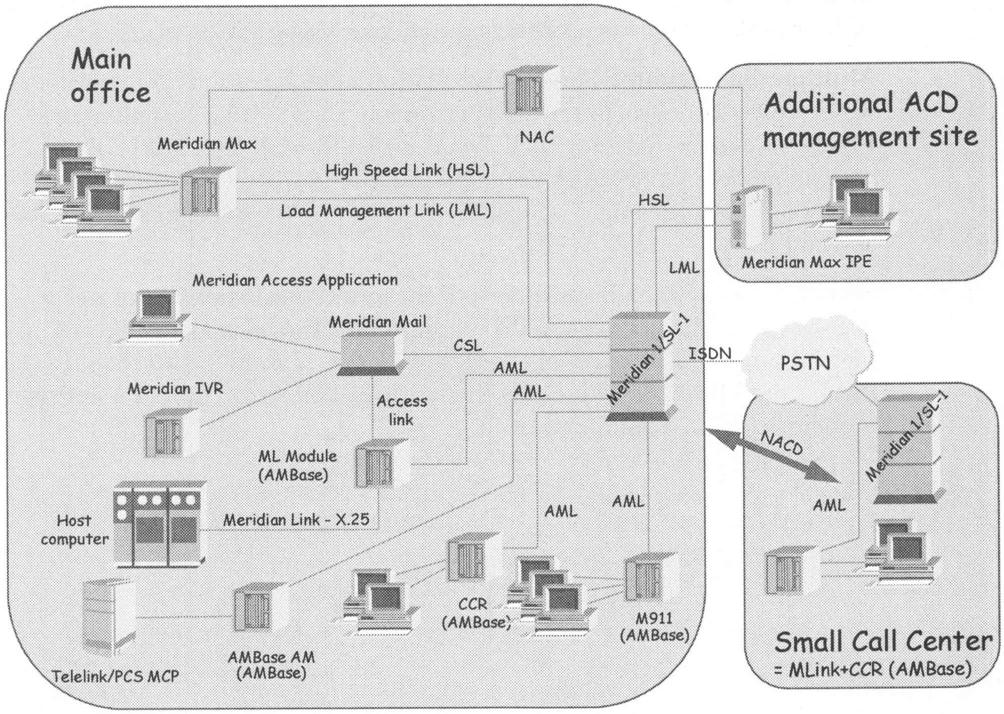
The Integrated Meridian Speech Directory (IMSD) was prototyped on the MicroNAV platform, which is a scaled-down version of Nortel's central office voice-activated services platform, the **Network Applications Vehicle** (NAV). The IMSD server comprises two IPE cards. The first one is a Service Provider Interface card (SPI) which connects to the IPE backplane and behaves like a digital line card; the second one is the Single Board Computer (SBC) which is Pentium 200 (or Pentium II) class industry standard PC motherboard. MicroNAV is an Ottawa box, based on the speech processing research done out of Montreal. IMSD will probably go to market on an NGen platform.

Multimedia Resource Server: Two platform configurations are available: the MRS/Tower connects to the high performance Intel/UNIX Applications Processor over an Ethernet or Token Ring LAN, while the MRS/AP—combining both Application Processor and Multimedia Resource Server functions in a single unit—is available as a low entry-cost solution for developers and end-users.

Several of the newer IPE cards also fit the model of off-board processing engines. The **Multi-Purpose ISDN Signaling Processor** (MISP) card provides Layer 2 and 3 of BRI (data link & network layer). The **S/T Interface Line Card** (SILC) and MISP talk with each other to make BRI work. Another example is mobility's EIMC, already discussed.

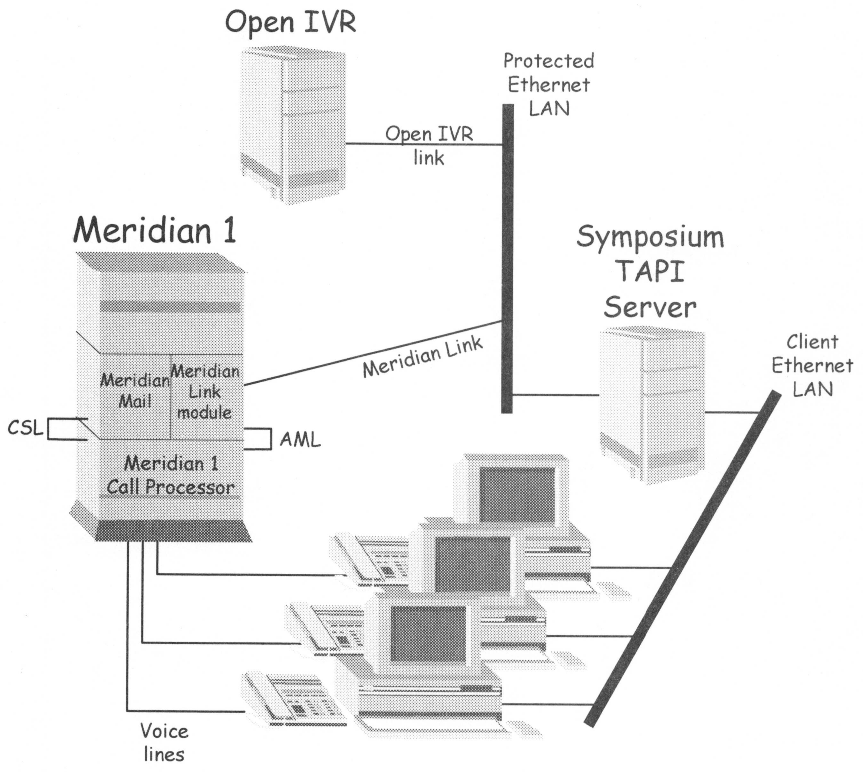
1.5.4 Putting it all together

The following diagram shows the kind of complexity typical of a big call center today. The proliferation of platform and link options should be simplified a bit under the NGen/Symposium regime, but it is equally likely that perfectly useful legacy systems will ensure that most sites will have a mixture of these and even competitors products to deliver their services.



(The agent positions shown would also naturally have M1 phones, but the diagram is already busy enough without attempting to show them.)

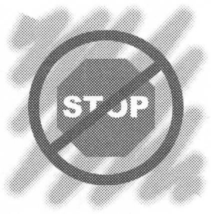
A more industry-standard variation of the same idea is to run Telephony Applications Program Interface (**TAPI**) over TCP/IP over Ethernet LAN. This is done using TAPI software on a Windows NT server, which connects by TCP/IP to Meridian Link, which in turn connects to the M1 using AML. Windows Clients talk to the NT server over the client LAN. This solution provides full Computer-Telephony Integration with no special wiring at the desktop for either phone or PC.



Notice that we require a separate, protected LAN to ensure that call traffic and customer LAN traffic don't interfere with each other. They may however be bridged together.

Part II

Implications for Design

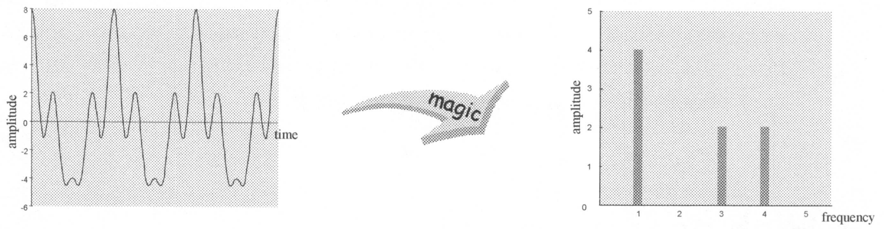


2. Pervasive design aspects

2.1 It's not our fault

Many M1 programming challenges can not be analyzed clearly at the "object" or procedural level because their solutions are smeared throughout the code. Perhaps surprisingly, this is not the fault of the designers or even the architecture; rather, it reflects the fundamental difficulty of isolating those aspects of the design that cut across a system's basic functionality.

A fairly good analogy from Digital Signal Processing goes something like this. When you look at a complex waveform, it can be hard to see what's going on. But if you could look at precisely the same signal in the frequency domain, it might turn out to be much easier to characterize, because behavior that was dispersed in the first view is localized in the second.



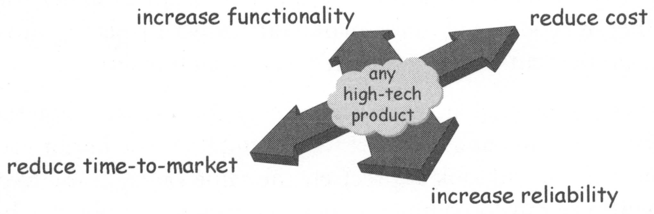
What is dispersed and therefore looks complex in the Time domain...

...may be revealed to be localized and simple in the Frequency domain

Similarly, the design aspects discussed in this chapter show up in many or all components, or exist in the interstitial spaces between components. Furthermore, they tend to apply at many levels of granularity. (Extending the above analogy, you get roughly the same spectral pattern regardless of your sample duration…) Trying to understand these aspects of a program by looking at its *structure* just makes the job more difficult, which may explain why they are not usually treated adequately in existing design specs or training courses. Therefore, this book separates the discussion of pervasive design aspects from the subsequent analysis of M1 software architecture.

Most of these requirements follow directly from the preceding chapter's discussion of the ways our customers deploy M1s. Software weenies will recognize many of them as flavors of classical good ideas for programming, with a bit of added emphasis on why they matter to PBXs. The tools that perform the coding magic analogous to DSP's Fourier transforms are only just starting to exist in research labs[8], and unfortunately are not available in the M1 environment at this time. Therefore getting it right will require conscious, consistent effort to successfully weave these considerations into a functional design. The degree to which we succeed in meeting these requirements determines the strength of our technical advantage in the market.

The other point to note is that some of the requirements we want to satisfy will pull us in opposite directions, and that how we choose to balance them is a strategic decision:



In the slow-moving days of government telecom monopolies and heavy regulation, Nortel did well to optimize around high reliability and feature richness. These days, our competitors are thriving by getting cheap stuff out the door fast, and we need to learn that this agility is important to our customers as they become less able to predict their needs.

2.2 High performance

We don't build Crays. We do neither heavy number crunching, nor high-volume data processing, and Pong had fancier graphics. Viewed in isolation, each transaction processed by an M1 tends to look pretty darned trivial. For example,

```
{phone goes off-hook → give dialtone}
```

So why would anyone call it high-performance computing?

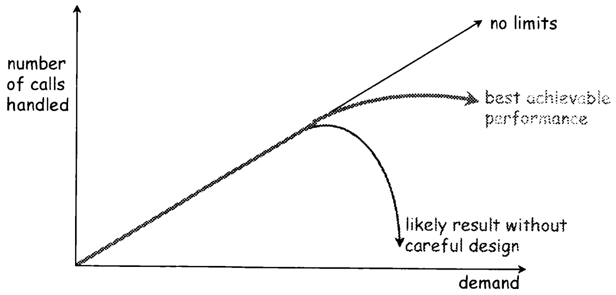
8 For an interesting discussion of a possible long-term solution to this problem, check out Gregor Kiczales' work on "Aspect-Oriented Programming" at Xerox PARC.

What makes the job tricky is that we must be able to handle a large number of these transactions with deterministic response times. The SL-1 architecture requires the main CP to handle every key press, display update, tone change, speech connection, and timer. Wink-start analog trunks must get a response within 70 milliseconds. We should supply a 3-second worst-case response to any telephone stimulus (and much faster is much better). On Option 11C, CP-controlled tones must have their cadences timed well enough to sound right to users. At the same time, the CP must be able to handle more than 250 transactions per second. Turning this around, on average we get only 4 milliseconds to process each transaction.

To start with, this means all code must be reasonably efficient. You'll see lots of design decisions that optimized for speed over clarity, reliability, etc. Provided these optimizations were not too extreme, they were probably helpful. Also, as will be discussed in the platform chapter, the scheduling algorithms we choose must give a bounded worst-case response.

Apart from the "hard" real-time requirements, there are a number of levels of softer requirements. To compete in the marketplace, we need to handle as many lines as we can for a given system cost. Craftsperson terminal response must be fast enough for efficient management of the system. Background maintenance and audit tasks must cycle frequently enough to keep our switches healthy.

Furthermore, when requests exceed our capacity limits, we need to do something sensible. We should keep processing as many calls as possible, and where possible apply some form of back-pressure to try to reduce the incoming message flow. We must not simply slow to a congested crawl under the heavy traffic (like shared-Ethernet does!). Getting this right takes fairly careful planning, and flow control at a number of levels.



And of course the payload streams (traditionally voice, and now video and data too) have extremely stringent real-time requirements. So far at least, these are handled in hardware.

2.2.1 No stopping (ever ever ever)

There is exactly one software task (`tSL1`) that handles *all* basic call processing. The time it takes to get through the main loop of this task affects both our latency and throughput, so it would seem obvious that we should never block execution within this loop.

Going to disk

Mobility software requires a database to keep track of nomadic subscribers. RogueWave provides a nice disk-based B-tree system, which seemed like an excellent place to keep this data. Unfortunately, disk reads and writes take time. It happened that an early version of the mobility software did this disk I/O from within `tSL1`, thereby holding up the rest of call processing every time a user wandered into range.

The overriding M1 design axiom is that `tSL1` doesn't stop. If you need to wait for something, or put time gaps between outgoing messages, or read from external devices, or anything like that, then either you have to go back into a call queue and wait for a new message to wake you up, or you need to have a separate server task to do the waiting around. And while any sort of blocking in `tSL1` messes up our ability to control latency, going to disk is particularly bad because disk accesses may fail for a variety of reasons, and if this happens from `tSL1` you can take down the whole switch. That is how the mobility problem was noticed…

2.2.2 Getting it wrong: Using the wrong algorithm

The order of performance of your algorithm, rather than its implementation, tends to define performance over a big data set. Even the best tactical optimizations will not make an O(n^2) algorithm faster than an O(n) algorithm if n gets at all big. We have to balance memory frugality against algorithm speed, but increasingly we should be willing to spend a few bytes to get faster code.

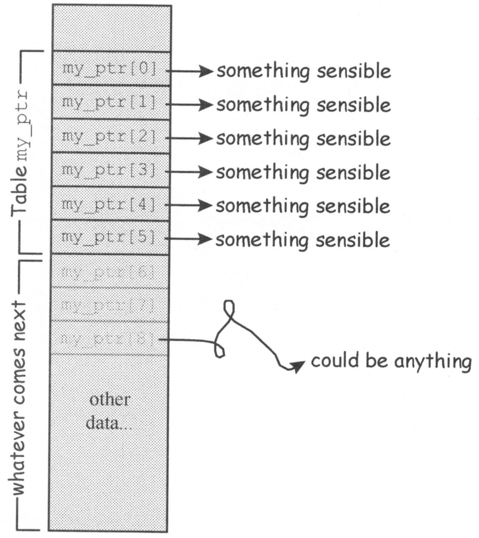
For example, until recently we used singly-linked queues for Call Register management. As of Release 21, we decided it made more sense to doubly link them in order to avoid having to rescan the entire list just to do a `BACK_PTR()` operation to find the previous entry in a queue. While this doubled the amount of memory dedicated to queue pointers, and marginally increased the work involved in most queue operations, it improved a mid-queue `dequeue()` operation from O(n) to O(1) performance. This was particularly important once we started having calls in multiple ACD queues. It also enabled us to rebuild the queues if we got lost.

Similarly, we have frequently chosen bit arrays to track service circuit allocation, and this requires scanning the array each time we need to find a free device. It probably makes sense to convert this code so that it uses a queuing strategy.

2.2.3 Getting it wrong: No seatbelts

Paradoxically, one way we can get the real-time challenge wrong is to take the wrong shortcuts. For instance, our compiler historically has no run-time checks for exceeding stack or array boundaries. This does speed up the code a bit. *But*, if we ever needed those checks, the resultant errors are extremely hard to track down.

Picture an array of 6 pointers. Now picture a piece of befuddled code that thinks the array is bigger than it is, and writes a number to memory using the 9th pointer in the array. Since the piece of memory it's dereferencing is probably not even a pointer, it could end up storing its number absolutely anywhere, and you wouldn't know it until something else needed the original value that got trampled. In fact, the errant code would probably work just fine, because when it went to read back its number, it would be right where it left it. This is one reason we use hardware memory protection.



2.2.4 Getting it wrong: Trying too hard

There are other examples in the code where we have just tried a little too hard to write blindingly fast code. Perhaps the most stunning of these is the core intrinsic for finding the data associated with a terminal, TNTRANS. We knew TNTRANS was going to get called a lot, and that it needed to pass several parameters into and out of the procedure. So we agreed that anybody who used it would first arrange a group of variables in memory in the following strict order:

```
INTEGER   xxx_ITEM;
UPOINTER [.U_TN_LINE] ULxxxPTR;
PPOINTER [.P_TN_LINE] PLxxxPTR;
UPOINTER [.U_TN_CARD] UCxxxPTR;
PPOINTER [.P_TN_CARD] PCxxxPTR;
UPOINTER [.U_TN_LOOP] ULPxxxPTR;
PPOINTER [.P_TN_LOOP] PLPxxxPTR;
PPOINTER [.P_TN_GROUP] PGRPxxxPTR;
```

Then, rather than taking the time to copy parameters to and from the stack, we only pass a reference to the first item. Then in the procedure code, we used the address of that item to find the addresses all of the others. Now this works, and it's probably faster than any alternatives, but it is *very* brittle code. Everyone just has to know that this is how you use TNTRANS. If anybody moves any of these variables, the code will break. If designers look for procedures that reference these variables, the cross-reference won't tell them about TNTRANS.

Most frustratingly, the existence of code like this means that we can not turn on certain levels of optimization in a new compiler that would have made *everything* a bit faster.

2.2.5 Getting it wrong: Not trying hard enough

The most common culprits here are code that has been written for non-real-time environments and ported to the M1. For example, when we first installed Orbix, we noticed that it was doing 17 `malloc()`s for every transaction.

2.3 Zero downtime

Perhaps the key difference between one of our systems and a typical PC is that the PBX is *always* in service. PC owners, despite their whining, have a remarkably high tolerance for system crashes (think 💣). Even healthy PCs are typically turned off periodically, giving them a chance to clean up many lurking problems. They may also get rebooted during maintenance, software installation, etc. Between planned and unplanned outages, a typical desktop PC is down for about 43 hours per year. In contrast, dependability is a performance requirement in telecom contracts, and our systems shoot for the industry standard of no more than 3 minutes per year out-of-service. Even this degree of unreliability might be a big deal: what if you have to make an emergency call (say, to your PC help desk ☺)?

Eric Raymond's thought-provoking paper *The Cathedral and the Bazaar*[9] bisects the world's software into (as you might expect) Cathedrals and Bazaars. He argues convincingly that the "bazaar" model of development, where diverse groups of coder-users hack into the source code almost at will, actually produces great software as long as you get a few key things right, the biggest of which is to release software early and *very* frequently. Given this perspective, and notwithstanding what I will state in Chapter 4, the M1 is a cathedral. This has historically been the right call, both because our distribution model couldn't handle more frequent releases, and because the overriding objective was for users to see as few bugs as possible—zero downtime. As our world changes, we should rethink this!

The single requirement for zero downtime bifurcates into two obvious design aspects: *high availability* (we're almost always able to handle calls) and *fault tolerance* (we more or less keep handling calls in the face of many hardware or software problems). It also implies a subtler one: no littering (because you can't rely on the next reboot to clean up after you).

2.3.1 Hot swap

If the switch is never down, the corollary is that changes must be performed while it is up. Fortunately, this is not as hard as changing the engine of a car while driving it, but it does take some planning. For hardware designers, this means cards and shelves must be able to be brought in and out of service with minimal impact to the rest of the system: "hot insertion" is the norm, and card enable/disable and diagnostic code must work on a live system.

At the software level, this has two different guises. The first is patching. We know that, despite our best efforts, there will always be bugs in the code we ship. Also, there may be special local customizations in some markets that we don't want to ship with the general release. And finally, there may be some features which we wish to sell to customers with older versions of the main software. For all of these reasons, the platform must support patching (adding complexity), but more than that, all code must itself be patchable, and this requires an

[9] It's on the web in at least four languages. Try http://www.earthspace.net/~esr/writings/cathedral-bazaar/.

understanding of the patching mechanism. In the pre-Thor days, when we still had to worry about things like non-resident overlays, there were more things that could make patching tricky. These days, about the only rule to remember is that patches work by changing where a procedure call branches to, so that an infinite loop *within* a procedure can't be patched without restarting the task.

The second version of software "hot swap" is that an entire new load must be deliverable to a running switch. Again, this adds complexity to the platform, and again requires some knowledge in the rest of the design community. For redundant systems (currently Options 61C and 81C), we load one half of the switch while the other side handles call processing, and then quickly switch activity over to the new side. There are three parts to loading the new side, which each require quite different processes. The executable code is compiled and shipped to the customer site. The protected data store is converted from the most recent version of the customer's local database by the system loader. And the unprotected data is built from scratch by restarting the switch. This strategy also gives us a back-out route if the new software fails spectacularly.

Hot software download is an extension of this philosophy. Some new components have dual program store banks, and Peripheral Software Download (PSDL) loads the new code into the standby side while the active side is processing calls. This means that we can do the download at a relatively leisurely pace (about 2 hours to load the biggest possible switch) because there is only a minimal impact to the customer—new features may not be available yet, and there is a slight decrease in traffic handling capacity.

2.3.2 No single points of failure

Even though modern solid-state hardware is very reliable, there should still be no single point of failure that can take down more than 400 lines[10]. Many of our critical systems have redundant hardware that would never be needed if the primary system always functioned normally, and this in itself leads to fairly complex platform software. Thankfully, most of this effort is encapsulated in the lower layers of code.

There's also a somewhat obscure French regulatory requirement that any redundant system must be able to gracefully switch over from one unit to the other, under normal traffic, with no more than 0.02% dropped calls. Since we

[10] Actually, this seems to be merely a good idea (as opposed to a regulatory requirement), but since we've spent the last twenty years telling our customers how important it is, it's probably here to stay.

want to sell globally, this puts some added pressure on our design teams, and rules out certain simple fail-over solutions.

The system must also be fault tolerant at all levels. For instance, if a peripheral fails and starts sending a torrent of spurious messages, or stops sending messages altogether, we need to isolate this error from the rest of the system. Calls involving terminals on that peripheral probably can not be saved, but they must be cleaned up gracefully, recovering all associated resources. And other calls should continue undisturbed. Similarly, most procedures, case statements, state machines, etc., need to do something quietly non-violent in response to an unexpected message. Flows should be designed so that bugs are only reported once in the chain of procedure calls, and this usually means not explicitly reporting when a procedure you call returns an error code. The best mechanism for doing this depends on where you are in the system, and so detailed examples of this are given in the Platform and SL-1 chapters.

Beyond all this robust hardware, it's crucial that our software doesn't take the system down. All of the code needs to be reliable, but extra-careful design, inspection, and testing are required for interrupt handlers, or code with data store protection or interrupts disabled. And to paraphrase Hoare, about the only way to guarantee correctness is to keep code so simple that there are obviously no defects. Keep as much as possible out of these critical segments.

2.3.3 No losing stuff

If things go well, we're always up. This means that if we have a memory leak (or any other resource gobbling) it's a big problem. If anybody allocates even one byte of memory as part of a regularly-executed task and then forgets to free it later, the system will eventually fail.

Memory fragmentation

A subtler problem that caught us when we jumped onto the Object-Oriented bandwagon was memory fragmentation. Typical OO practice makes heavy use of "the heap" (an apt description for the amount of planning and organization involved in this memory management strategy). Whenever you need to work with some data, you allocate a work space on the heap, do your stuff, and then free it later. This is a really bad idea in our environment, for two principal reasons.

The first is that it costs real-time. It's always preferable to do as much work as possible at compile or loadbuild time, rather than during call processing. Generic routines to allocate and free a block may take a fair bit of time.

The second is that in the general case this technique will leave us with a bunch of small, non-contiguous pieces of free memory. Non-real-time OO systems detect and correct this problem with a technique called "garbage collection". GC algorithms are improving, but they still cost a lot of real-time, so we currently don't attempt to do any. In the absence of garbage collection, we just end up with a bunch of increasingly-small free store fragments, and progressively slower `mallocs`.

So when's the best time to allocate stuff? compile or load time ☺ > init time 😐 > run time ☹

2.4 Nobody's manning the switch

Many sites, especially those with small PBXs or buildings where the M1 is acting as a remote concentrator for another main switch, have nobody watching the system. Who can blame them? With an expected 3 minutes per year total downtime, it would be worse than being a Maytag repairman. The effect of this is that our systems have to prevent, detect and correct their own problems.

2.4.1 An ounce of prevention: Protected memory

Some industry experts argue strongly against the need for protected memory. The claim is that it adds complexity and cost to the hardware design (true) and that it's slower to write to protected memory (also true). Nevertheless, we do use it for two main reasons.

The first is that, even now that we're using VxWorks, we have a single flat address space. Any task can read from or write to any address in memory. We use hardware-enforced memory protection to prevent bad code from corrupting critical data. If somebody forms a bad pointer and tries to use it to write to protected data, their process traps. Note that you can still trample unprotected data, and this does lead to horribly difficult debugging problems. (*Hint: avoid any temptation to do pointer arithmetic.*)

The second reason we use protected memory is that, by protecting the more stable data, we significantly improve the odds that a warm restart will successfully bring us back to a healthy state. "Checkpointing" is a standard fault-tolerance technique whereby the current state of a program and its data, including intermediate results, are saved to some non-volatile store. If the program gets lost, it can be restarted at the point at which the last checkpoint occurred. Checkpointing our system, with thousands of calls that are all essentially independent processes, is impractical. Our variation of this is to keep stable data in memory that has hardware protection against inadvertent changes. This really had a lot more benefit in the days when it took 15 minutes to load the whole image from a tape drive. These days, there is no time advantage to warm restarts over cold ones. The benefits are that we don't lose protected data (like recent database changes) and that we preserve those calls that are already talking.

A further margin of safety is provided by saving some data to a disk or flash ROM. This is done by hand-building the required code to save the "important" bits. In particular, for historical reasons the M1 disk memory is 16 bits wide, and so no attempt is made to store the *structure* of things like linked lists and DN trees. This turns out to be helpful when upgrading from one release to another, where internal data structures are highly likely to change, and it also saves disk space. The downside is that it's more stuff to get right (which means that it sometimes goes wrong); we can't just set a flag saying "this data should be persistent across power failures and reboots".

See Chapter 5 for a more complete discussion of restarts and memory types.

Safe languages

A "safe" language protects untrustworthy programs from each other, and might allow us to abandon hardware-based data protection. The key idea is that object references (pointers) are unforgeable. Pointer authenticity might even be combined with type-of-access permission (for read, change, and delete). Such a language would need to have at least the following characteristics:

- all pointers are automatically initialized, at least to `NIL`
- pointer arithmetic is disallowed
- re-casting pointers (to override type checking) is disallowed
- array bounds and stack overflow/underflow checking is enforced

Since *all* of the resident M1 languages give you more than enough rope to hang yourself, it is unlikely that we will abandon memory protection soon.

Besides buzzword compliance, one reason we might consider porting the M1 code to Java would be to achieve this safety. However, now that our customers believe that hardware data protection is buying them some reliability, we might conceivably choose to keep it, if only for marketing reasons…

2.4.2 Watchdogs

How do you tell if you're deadlocked, in an infinite loop, or in some other way failing to get the job done? You can't detect all cases automatically, but we catch most by running "watchdog" timers. When software is functioning normally, it periodically resets this timer. But if the timer ever expires, it automatically causes some kind of reinitialization. Watchdogs will be discussed in some detail in the Platform chapter, but it is worth remembering that all software must complete its processing within the watchdog time window, and that every SL-1 service must provide suitable code in module `INIT` to be invoked during a restart, for example if a watchdog timer does expire.

2.4.3 Super-ultra-mega-reliable restart code

A barking watchdog, a dying task, a hardware fault, or a craftsperson's command can all cause the system to reinitialize. This will cause an outage, but the hope is that it will also clean up some existing problem and lead to a healthier system. If an outage is bad, failing to recover is even worse. If critical sections of code must be good, initialization code must be perfect. By the same logic, the code that saves and restores the customer database must also be very clean. We also need to think up ways of making initialization code better than standard code at finding and fixing (or at least tolerating) errors, since the fact that init code is being invoked suggests that there may be something wrong with the system.

Call processing code presents a special challenge after a warm restart. Call Registers, which hold all of the detailed data describing a call are stored in unprotected data. But unprotected data is cleared during a warm start (although calls are *not* taken down). We achieve this alchemy by examining the Network Equipment connection data, and reconstructing the CRs based on what we find there. For simple POTS calls, this presents no real difficulties (although we lose those calls that were not in a stable talking state.) For featured calls, the connection memory is insufficient to rebuild all the data. Usually, terminal states are reset, CDR data is lost, and all references to the feature are cleared, so it is both impossible and unnecessary for the feature code to do anything at this point. Still, it leads to some possibly nasty failure cases, especially for networked features, and all software should have a plan for approximately graceful failure.

As mentioned in Chapter 1, call centers hate losing callers. And as just mentioned above, we normally drop calls that are not in a stable talking state over any restart. Unhappily, a busy call center tends to have lots of interesting calls waiting in the queue, so we've always had some pressure to try to keep from dropping them. The problem is, we don't have any spare real-time on these switches, so we can't build the ACD queues in protected memory (even if the idea didn't scare the bejeezus out of us—queue manipulation is more prone to corrupting memory than most code, but at least it's usually unprotected memory so it might get cleaned up by a restart). We recently managed some slight-of-hand to rebuild the queues, but the solution is probably a bad one for general use: we sneak some of the queue information into a part of Thor's memory that SL-1 does not generally know about, so `INITIALIZE` doesn't try to reinitialize it. This solution works, and it's fast, but it has already caused us its first problems.

2.4.4 The backup plan

In a good year, we have zero downtime. Failing that, even if something very bad happens, a restart will almost always clean up the problems. But even more severe corruptions may be fixable by reloading the database from disk or flash memory. `INITIALIZE` is then called, which does an especially thorough job of setting up all the critical data structures. This of course will cause an even longer outage than a regular restart, but with a bit of luck it will leave you in a known, healthy state.

Sometimes even rebooting won't help

One Rls22 site had an 8-hour complete outage that could have been avoided by better system awareness.

The Story: We save historical alarm data on disk (ironically, part of the effort to improve reliability). Somehow, we managed to corrupt this file. On the next trap, when we tried to open it, the event collector task choked. This killed the restart, and another one was immediately started. Because the corruption was on the disk, neither reinitializing nor rebooting could improve the situation.

Moral #1: The more permanent the data, the more important it is to avoid corrupting it.

Moral #2: Since a serious bug in *any* restart code will knock out all call processing, it's best to keep the restart code as simple as possible, and defer non-critical work until after the restart.

Moral #3: It's a good general rule that a less important job should never hold up a more important job. In this case, we held up *everything* because one disk file was hosed. Even pretty thorough testing is unlikely to detect this sort of problem. Only careful design will save you.

2.5 Customers' budget worries

Although the average cost per line over the lifetime of our systems is low, the initial purchase of any PBX tends to be a large capital investment, and subject to more intense scrutiny than, say, that of a desktop PC. Customers have to believe that they are investing in a product that will continue to be useful for a long time. At the same time, even the rare customer who thinks they know exactly what system they will need over the next ten years probably can't afford to install the whole thing on the first day.

"no forklifts"

Capitalizing on customers' fears, PBX vendors regularly describe each other's system changes as "forklift upgrades": customers will have to cart away their installed box, with enormous cost and disruption. A big selling feature of M1 is our "evergreen by design" story. This line helps us convince a skittish customer that (a) their system won't be obsolete tomorrow, and (b) they can upgrade it gradually as their business grows or their needs become more complex. Their initial investments will still look sound, and will continue to amortize out nicely over the coming years. Their incremental needs will be met by only incremental investments. And the cutover from current equipment to the next generation can be achieved with no noticeable outage.

2.5.1 Scalability

The first design implication of the evergreen policy is that M1 capacity must have a continuous upgrade path from a small switch (overlapping with the high end of the Norstar product line) to a fairly large one (overlapping with a low end SL-100). This is one of the big reasons for having a line of modular "Options", rather than simply having several different switches of varying capacities. If you need more terminals, add a PE shelf. If you need more simultaneous calls, add an NE shelf, or maybe upgrade your CP. Even the Option 11C can add extra cards and extension cabinets.

Pivotally, the substantial costs of configuration, installation, testing, and user education (which combine to be greater than the cost of the switch) do not need to be repeated. For instance, even if you have to upgrade your CP, you can probably leave all of your existing network and peripheral equipment, cabling, and terminals exactly where they are.

Even a customer buying a very big switch must believe that, if it were necessary, the switch could handle even more traffic, or at worst that we will soon announce an upgrade that could. This is mostly a hardware requirement, but handling the transitions shows up in software, particularly in platform and maintenance code. Also, because this has caused the M1 to be a hierarchical multi-shelf cluster of components, the M1 itself becomes a network, and subject to most of the potential problems that occur in networks more generally.

2.5.2 Software packaging

Packaging fills two different needs: it allows us to keep loads to a reasonably small size (much more true historically than today) and it allows us to charge customers incrementally for added functionality. The mechanism has changed through the years, as the surrounding technology and cost trade-offs have shifted, but the concept of having easily-turn-off-and-on-able code seems likely to persist for a long time.

This shows up in the software in two ways. The first effect is that the platform must provide the control mechanism for installing and activating packages, and the packages themselves must have the corresponding hooks to allow that activation. The second, more subtle effect is that that one package can not in the general case assume the existence of others.

2.5.3 Configurability

Not all customers want the same things. Even two customers who want the same feature set may well need them to behave quite differently from each other, so having configuration data to control this behavior is sensible. However, it is equally important that the switch can be configured with a minimal amount of effort, and therefore it is critical that we reduce optionality to that which customers need, and give sensible defaults for each option.

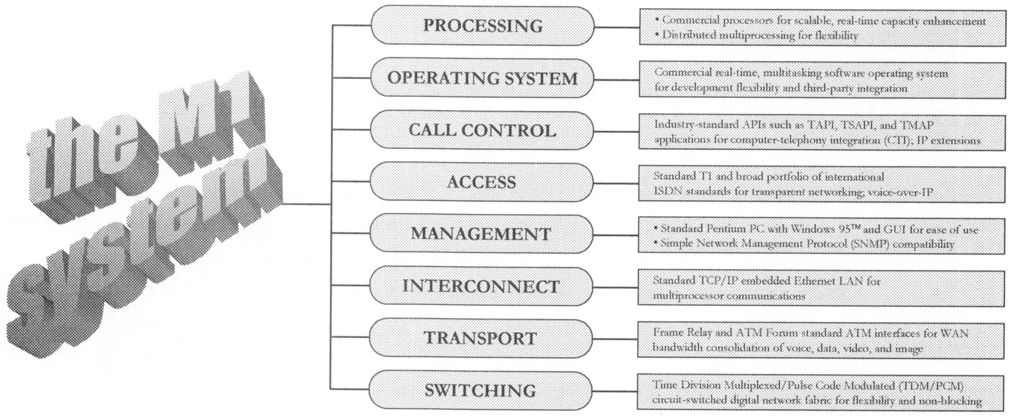
Configurability allows system costs to match customer value more closely; the cost contributed by functions a given customer doesn't need (perhaps Mobility, for example) must be minimized.

In early SL-1 software, the `SET ORIGIN` statement was frequently used to map software explicitly to given memory locations. Over time, we moved this decision into later stages of the loadbuild process, and then out into restarts or even run time. Done carelessly, this risks memory fragmentation and real-time problems, but is does add a degree of flexibility that reduces design and engineering effort.

2.5.4 Modular design

In addition to a clearly modular hardware design, we have built some degree of modularity into our software. The following is the ***marketing*** view of our software structure. It does *not* correspond precisely to the real hardware or software structure (described in Part III of this book), but it is still important in that it represents what our customers believe to be an important aspect of their investment. The theory is that each of these encapsulated layers may be upgraded independently of the others, and this is approximately true.

Arguably, there should be a layer in this picture called "Terminals", but we didn't really separate out a clean terminal abstraction. About the closest we come to this is to reuse the SL-1 set interface to drive other terminal types, and this has resulted in some slightly mutant code.


the M1 system
PROCESSING
• Commercial processors for scalable, real-time capacity enhancement
• Distributed multiprocessing for flexibility
OPERATING SYSTEM
Commercial real-time, multitasking software operating system for development flexibility and third-party integration
CALL CONTROL
Industry-standard APIs such as TAPI, TSAPI, and TMAP applications for computer-telephony integration (CTI); IP extensions
ACCESS
Standard T1 and broad portfolio of international ISDN standards for transparent networking; voice-over-IP
MANAGEMENT
• Standard Pentium PC with Windows 95™ and GUI for ease of use
• Simple Network Management Protocol (SNMP) compatibility
INTERCONNECT
Standard TCP/IP embedded Ethernet LAN for multiprocessor communications
TRANSPORT
Frame Relay and ATM Forum standard ATM interfaces for WAN bandwidth consolidation of voice, data, video, and image
SWITCHING
Time Division Multiplexed/Pulse Code Modulated (TDM/PCM) circuit-switched digital network fabric for flexibility and non-blocking

2.5.5 Reuse

Modularity leads very directly to reusability. In the hardware domain, it means we can use IPE with various generations of CPU technology, and even with the SL-100. Similarly, new terminals, protocols, processors, and features are inevitable, and a modular design helps ease the integration effort.

In software, it allows us to evolve each component in isolation. This greatly improves the effectiveness of multi-site development. It also helps shorten our time-to-market, which is crucial in our business.

While some of the original rush of enthusiasm in the software community about the value of software reuse has been tempered by actual experience, it still seems likely that some reuse is a good idea. As with performance aspects of our design, the wise advice is probably to strive for the Buddhist ideal of the "middle path".

Dubious reuse example

Maybe we shouldn't always attempt reuse. There *is* reuse, but its kind of scary, in Flexible Feature Code. Basically, you go down the DN tree, terminate on an FFC flag, strip off and store the FFC digits, replace them with the Special Prefix (SPRE) fixed feature access digits, and retranslate. (This is only true for features that were written pre-FFC.) It works, but it adds complexity to the SPRE code, and it costs us real-time. Is it good news? It's not entirely clear, really.

2.6 Bigness

2.6.1 Vestigial code

Once the investment has been made in a huge code bulk, nobody can bear to throw it out. So we usually end up making only incremental changes to our legacy product with each release. At one level, this is just reuse, and is often a good idea. It always appears to save us money, when compared to the cost of rewriting the whole thing from scratch. But it can still be very expensive in other ways – the resultant code is likely to be bigger, slower, and clumsier than what we might be able to achieve by starting from scratch. At some point, these costs outweigh the initial savings.

We also suffer from the well-known "shared refrigerator syndrome": many people put stuff in, and nobody wants to remove anything that might still be wanted. Things get duplicated, people move, and projects get canceled. Before long, it's not a pretty sight. A few years back, a static analysis of the SL-1 code showed that 8% of the total volume of code was *completely unreachable.* Nobody called it. Ever. This did not even count the intentionally cautious code which could only be reached under perverse error conditions, nor places which were logically unreachable because of tautological comparisons. These were simply procedures which only had definitions, with no invocations. That's over 150,000 lines of useless flab. And even after this analysis was done, an executive decision was made to leave it all there, just in case it was somehow needed after all.[11] This problem is certainly exacerbated by the lack of a clear software ownership model.

At the same time, like the bachelors' fridge with five half-full bottles of ketchup, we have several known instances of useful code with unnecessary replication. On the trivial end, the analysis just mentioned noticed nine different algorithms, with about twenty instances each, of code to write "-------" to the telephone display. eg:

- `for i:=1 to 7; output "-"`
- `output "----"; output "---"`
- `for i:=1 to .SEVEN_DASHES; output "-"`

This may seem unimportant. Okay, at this scale it probably is unimportant (since you might take as much time finding and testing a utility to do this as writing your own), but we have much bolder examples as well…

[11] Reliable rumor has it that most of this code was subsequently liposuctioned as a skunkworks project, but the mindset that was afraid of touching it is revealing. There's a fairly deep-rooted fear of the SL-1 library in some corners of the development community.

Name transmutations

One of the first things a new designer notices here is that we pick an internal "working title" for most projects, and then let marketeers rename the technology when we're ready to sell it. Cybele becomes Meridian Evolution. TNT becomes DMT becomes Pangaea becomes DTEV. You just have to know that they're all exactly the same thing.

Meridian mythology includes a (possibly apocryphal) tale of how this once got us into trouble. Once upon a time, we renamed a terminal project some time during the development phase. Unfortunately, the integration team was passed the code with each name, and somehow integrated it twice: once with the old name and then a second time with the new name. Great battles are then said to have been waged over the need or difficulty of backing out the original copy, and further rumor has it that both are still there today.

Some SL-1 storytellers insist that these events never happened, but to do so is to miss the point of fables. We should think about why our development process makes the story merely unlikely, rather than implausible…

2.6.2 Data hiding

Perhaps the central strength of OO design is that it codifies the not-so-new notion of data hiding. An earlier way of thinking nearly the same thought was called "Abstract Data Types", and it too taught that you want to hide the data for your service beneath some sort of API. People can then access it only through the procedures you provide, and shouldn't know or care how you've stored it.[12] This was a good idea in the '70s, and it's still a good idea today, although then as now there are engineering considerations that sometimes force other choices in embedded systems.

A narrow interface (with implementation details hidden) also fosters reuse. It makes it much easier to identify the exact behavior of a piece of code; it means you spend less time trying to figure out the interface (because it's smaller); it allows code on one side of the interface to be replaced while reusing the other side; and it makes the code more likely to be trustworthy because you can see all the ways it might hurt you.

[12] The canonical example is the "point" data type provided by a graphics package, which might be stored internally in either polar or rectangular coordinates, or even something more exotic, without affecting its usability by client applications. This example also shows the practical limitations of the technique: the choice of coordinate system will impact how easy it is to design certain functions, how quickly they will execute, and, with the discreet mathematics of computers, maybe even the results. But at least with an ADT, you can change your mind, change the implementation, and leave the rest of the program intact. Add inheritance and polymorphism to ADT's, and you've got most of OO.

Unfortunately, the heavy use of the single shared data interface POOL, while it does make for some speedy code, runs precisely counter to this advice. Where you have the opportunity, minimize what you put into the global pool by using local COMPOOLs and nested procedures. Other subsystems should only be able to see the things they need to.

2.6.3 More about reliability

We don't have a protected operating system. If we did, we might be able to claim that an error in one modular software component would be *unable* to adversely affect any other components. Instead, we have to design carefully to try to minimize this risk. This becomes especially important as we start to pull third-party stuff into the system, either off-board (CTI, SNMP, billing systems) or—scarier yet—on-board (Orbix, Envoy, Seaweed, etc.) The sheer scale of the system makes this problem especially formidable.

In the early 14th Century, a Franciscan monk named William of Ockham wrote:

Entia non sunt multiplicanda praeter necessitatem.

Since fewer and fewer software designers speak fluent Latin, this is now usually studied in translation: "Keep it simple, stupid". Keep extra features from creeping into your design. Factor out recurring code into a single place. Minimize interfaces. Resist the temptation to add complexity to what is already a sufficiently complicated system. Less is more.

2.6.4 Paranoid code

It's not too hard to find "redundant" SL-1 code. For example, there are ELSE clauses started by comments like "`% we should never get here`", and DEFAULT clauses in complete-looking CASE statements. They're in there because the system is too huge to keep perfect. This defensive driving does make the system bigger, and correspondingly more difficult to understand. It's still usually the right thing to do, so that at least you get predictable failure modes when bad things happen.

2.7 Networking complications

2.7.1 Standards compliance

An M1 is usually a leaf node in the huge global telephone network, and therefore the standards-compliance of protocols, and even of feature operation, becomes important. This means we sometimes have to build new interfaces in a way that isn't very natural for our existing system. This in turn means it may be worth our effort to influence standards directions.

2.7.2 Version control

Because different nodes are upgraded at different times, we need to support many versions of the various protocols simultaneously. This usually requires being able to support versions that are both newer and older than the one you would prefer to work with, at least for a subset of your messages, and hopefully at least rudimentary success at handling fairly different implementations from competitors' products.

This problem pops up at many different levels. Now that we've put intelligence in the peripherals, these may get out of sync with the CP. We also have to agree on which versions of which protocols we will use to connect the M1 to the PSTN. Application Processors and management stations may have the most sophisticated protocols, but at least they tend to be industry standards like SNMP or TAPI. The Meridian Administration Tools (MAT) product has a "release" field in its common database for each switch that it connects to, which all its applications reference. When asked to update its system data, MAT queries each switch to establish what software is currently loaded. It can simultaneously support PBXs running different loads.

It appears to be a good rule of thumb to put a standard interface between components that you expect to want to upgrade independently, even if we currently control both ends of the pipe.

2.7.3 Glare

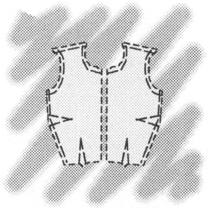
Finally, a standard network call processing headache is "glare". This term describes the situation when both ends of a trunk are seized simultaneously. It can happen on two-way trunks because each end is being driven by an independent switch. You can engineer around it, but only by having dedicated incoming and outgoing trunks, which is not generally an acceptable expense. Trunk call processing code has had to be set up since day one to be able to detect and handle this problem. Glare is really just a special case of the generic problem of having a bunch of free-running computers in the network, of which the M1 core is only one, all of which are trying to update state information. All designs should think about this.

2.8 Conclusion: PBX software *is* rocket science

PBX software turns out to encounter most of the tough problems in computer science simultaneously. In particular, we worry about:

- real-time system problems
- high availability, with some fault tolerance
- classical distributed applications problems
- multinational, multivendor protocol and feature interworking
- multisite development, including third-party code
- a large, complex legacy base, with an evergreen support model, and upgrades expected to have zero outage time
- two orders of magnitude differences in scale between various sites
- cost

About all we're missing is heavy number crunching. To nobody's surprise, doing all this consistently is kind of tough. We can't afford to ignore tools and processes that might increase our chances of getting it right.



3. Patterns

What's a "Pattern"?

Feel free to skip this box if you already know why "Pattern" is the Object-Oriented community's word-of-the-month…

The study of patterns is a widespread attempt to start describing standard, effective solutions in OO design. It is today's equivalent to the efforts of our forefathers to catalogue various sound algorithms for building stacks, sorting data, etc. that were distilled in Knuth's classic cookbook, *The Art of Computer Programming.*

At about the same time that Knuth was pondering linked lists, a real-live architect, Christopher Alexander, was doing what architects do: thinking about designing buildings. His thoughts resulted in *The Timeless Way of Building*, which proposed a pattern language for architecting buildings and cities:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

The idea struck a chord with the Gang of Four, who had been plagued by a vague *déjà-vu* each time they solved a common design problem, and asked the obvious question, "Isn't there a pattern here?" *Design Patterns*, their seminal book, was their answer.

Then last year, five engineers at Siemens wrote *Pattern-Oriented Software Architecture*, which not only is trendy but also seems to be helpful for thinking about how we design large systems. Apart from providing yet another catalog of reusable Patterns, perhaps their most helpful contribution to the field (and the one which I'm about to reuse in the true spirit of patterns) was an organizing principle for understanding when to use particular patterns:

- **Architectural Patterns** provide the skeletons for system-level architectures.
- **Design Patterns** are strategies for handling component structure.
- **Idioms** are tactical plans for implementing a design given the idiosyncrasies of the language & programming environment we're using.

"But wait!" you clamor. "*Patterns* may be a great OO word, but the SL-1 is very non-OO code."

Okay, this is definitely true, and for that reason, many canonical OO patterns don't show up in SL-1. This chapter is included for two reasons. First, the OO patterns do (or at least probably should) apply to all of the new C++ code. By including this chapter, I get a convenient place to ask designers who are serious about writing good C++ to go look at some of the Pattern literature, starting with http://hillside.net/patterns/patterns.html. Nortel also now has a "Pattern Center", with a home page at http://seal/NPC.

The second reason is that, although the Pattern movement happens to have grown out of the OO community, there's no particular reason the thinking should only apply to OO code. I claim there are some clear patterns in the SL-1 code. This should not be surprising, since all of the pervasive requirements just discussed in Chapter 2 keep pushing people down similar paths as they try to solve similar problems. Understanding SL-1 patterns should be as valuable to SL-1 designers as understanding OO patterns is to OO designers. And the less idiomatic patterns are probably valid in both solution spaces.

I'll even add in a new twist. Like the other Pattern books, I'll catalog some good patterns (👍). But I'll also describe some common bad patterns (👎), in the hope of helping people understand imperfect designs without encouraging their replication. And (succumbing to temptation) I'll even describe some things that frequently go wrong, with a list of ugly patterns (☠). Knowing about the ugly patterns may at least reduce debugging time, but should also help people think about ways to avoid the problems in the first place.

Following tradition, each pattern will be described in four steps: a **name** for it, a **context** it appears in, the **problem** it is solving, and the **solution** itself. Real examples from the M1 are also included.

Good patterns tend to look obvious to experienced designers. After all, the whole point is that they've been seen before. Nonetheless, they provide a good shorthand for discussion, can reduce time-to-market by encouraging *design* reuse, and aid new people's understanding by exposing the wisdom gained over decades of building telecom systems.

3.1 Good patterns

3.1.1 Transaction engine

Context: This is ***the*** big architectural pattern in PBXs (and many other telecom products). It is the single key to understanding a big chunk of the SL-1 software. It forms the backbone for all call processing and administration. You can use this pattern whenever you need to maintain the integrity of a complex object model in the face of asynchronous change requests from multiple agents.

Problem: The bulk of the work a PBX does for each terminal could be specified by a Finite State Machine: get a request, perform an action, move to a new state, and wait for the next stimulus. The changes to state information normally affect how subsequent transactions should be processed. For example, if one transaction attempts to terminate a call on a given phone, and another transaction attempts to originate a call from that same phone, then the order in which they are processed has a great bearing on who ends up talking to whom.

Because you have to service requests from a large number of autonomous agents, you can't lock the database and wait for the first call to finish. The transaction requests individually have implicit real-time response requirements (at least soft ones), and the capacity of the overall system must also be maximized. But even if this weren't true, you might cause deadlocks if two agents were involved in related calls.

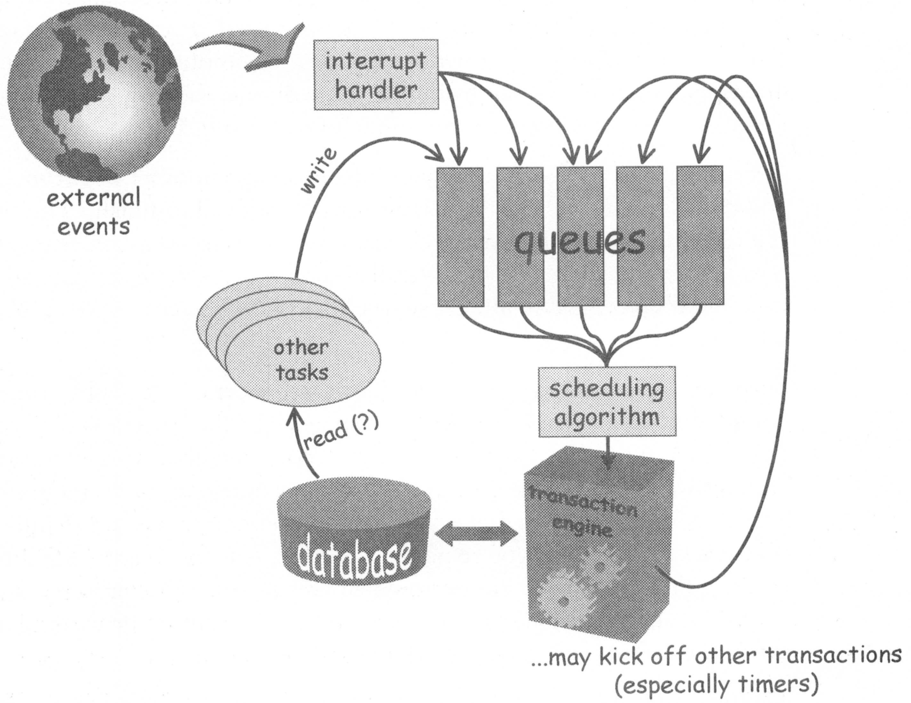
Solution: The best way to handle all of these worries is usually a run-to-completion Transaction Engine. The essence of the pattern is that you handle exactly one transaction at a time, and you complete that whole transaction atomically, as fast as you can. Interrupts may queue up other requests while you're working, but you don't stop working on one transaction until you have performed any actions you're going to perform, changed any state information you're going to change, and returned any result you're going to return. This pattern guarantees that, except for the call you're actively processing, every other call in the system is in a known, stable state.

The broad term M1 software uses for the state information that is saved between transactions is Progress Mark (PM). Suspect a Transaction Engine any time you run across a variable named `anythingPM`. For call processing, the place the known, stable call state information is kept is generally the Call Register (CR), and particularly `MAINPM` and `AUXPM`. For service changes (overlay processing), look for the global variable `SCSTEPPM`. *Many* other PMs exist.

The fact that each transaction must be *completed* in a way that lets the database return to a stable dormant state will guide the structure of your transactions, and what should constitute an event. By convention, the parameter passed to a transaction handler that specifies which event has happened is called SOURCE on M1. See module POOL for about 150 examples of SOURCE event lists.

If absolutely necessary, other tasks may read from the database without going through the transaction engine, but they must understand that they risk having different pieces of data misaligned if they interrupt the main transaction engine during an update.

Besides maintaining data integrity, this pattern also prevents the sort of deadlocks that happen whereby two processes end up waiting for each other.



Real examples: Recognizing this pattern is tricky because the actual *code* tends not to look exactly like this pattern. The server side of a client-server design almost insists on a Transaction Engine model, but there are also bigger but more subtle examples in M1. The biggest is the one that runs the entire call processing database: WORKSHED. No task that supports more than one terminal can wait around for input. But if you had a different task to service each terminal, you would eat up enormous amounts of stack space, risk new race conditions (see

"The usual suspects", below), and take the real-time hit of a context switch for every incoming message. What really happens is that, rather than having 1000 tasks to handle 1000 calls, WORKSHED does the waiting on behalf of everyone, like this:

```
WHILE .TRUE DO
   poll all input & timing queues in order of priority
   IF any queue has a transaction waiting
   THEN
      process transaction
   ELSE
      do background work
   update timing queues
```

There's also nearly an exact duplicate in the overlay manager, which is one of the jobs scheduled by WORKSHED.

This is a pretty standard sort of idea with multi-user databases. The trick is in seeing a PBX as just a big collection of multi-user databases. One database is the collection of active calls. Another is the equipment configuration. Another the directory numbers. The ACD queues might be viewed as yet another. Unfortunately, a change in any one of these may affect the behavior of others, so we need to treat the entire mass as the single "everything call processing needs to know about" database.

There are various things to get wrong, especially if you don't understand the pattern.

- ✘ Note that in a hard real-time environment, the best latency we can really promise is equal to the execution time of our longest transaction. Therefore, we need to limit the work done for any given transaction. The period allotted to process a transaction is called a "timeslice", and is usually capped by the platform. On the DMS-100, any process that exceeds its timeslice is killed, and the call cleaned up. On M1, we can't detect an individual transaction running long (usually an infinite loop or deadlock) until the watchdog timer expires (after 2 seconds). In this case, we actually restart the entire system.
- ✘ If you suspend in an unstable state, with half a transaction processed, things you thought you knew about the switch, like which phones are on-hook, are likely to change while you sleep, and you may disrupt other software that depends upon stable dormant states.
- ✘ Unless you're very careful, this pattern is easily violated by priority-based multitasking. It is crucial that no higher-priority task can alter the state of your data. This is a good reason to make sure that only the SL-1 task can change call processing variables.

✘ It's even harder to get this pattern right if you try to share processing across a number of CPUs. This will be the central challenge if we ever attempt to evolve to load-sharing symmetric multiprocessing (ie: add more CPUs whenever you need more power).

✘ If you forget to update your Progress Mark, you're in an infinite loop. Stop snickering! This happens all the time. Fortunately, it's usually quickly detected during the test phase. In SL-1, the usual cure is to call a local procedure called something like `INCRSTEP`.

3.1.2 Layers 👍

Context: You can use Layers to construct any large system that needs decomposition. This is Architectural Pattern #1 from *Pattern-Oriented Software Architecture*. A brief description follows, but you can refer to that book for a really good discussion of how to build Layers right. Good layering has the added bonuses of increasing the odds of code reuse and the ease of portability.

Problem: You need to be able to think about, and then build, a complex system or subsystem. You want to allow a bunch of designers to work simultaneously on the problem. You may even want to buy parts of the solution. You want to provide as much flexibility as you can afford to put in, because platforms and interfaces are likely to evolve. Performance is an issue.

Solution: Design your architecture as a series of layers. Probably start with the lowest layer of abstraction, and work your way up. Ideally, each layer should build its services only upon those provided by the layer immediately beneath it, and all of its components should be at the same level of abstraction. Each layer provides a fairly narrow API to the layer above it, and its implementation may be changed extensively provided that the API is retained.

This pattern may really just be a guideline for implementing Dijkstra's separation of concerns, "to deal with the difficulties, the obligations, the desires, and the constraints one by one … to reduce the detailed reasoning needed to a doable amount"[13].

[13] The long version of this discussion appears in *A Discipline of Programming*, Dijkstra, 1976.

Real examples: The most conspicuous example of Layers in telephony is the OSI 7-layer stack for networking protocols. You know:

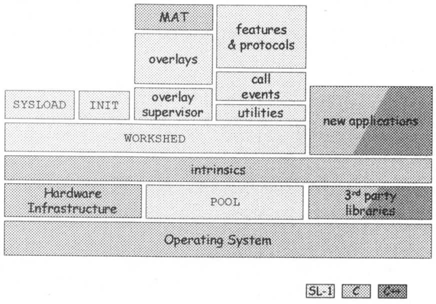
application
presentation
session
transport
network
data link
physical

Both the spec and the usual implementation of this stack embody Layers. This design's choice of seven layers is at least fortuitous, if not deliberate. Psychologists tell us that we can really only keep track of 7±2 things at a time in our short-term memory. Not exceeding this limit to the number of layers helps designers to keep their brains around your design.

Cautionary tale

If somebody goes to the trouble of designing a layered architecture, it's important to respect it. At least one ISDN feature on M1 violated the orthodox ISO layering. It was written on top of PRI layer 1 & 2 by calling the routines directly, rather than on top of the ISDN service. There was a bug in this feature. Our storyteller tried to debug it in the field by using the ISDN trace utility, which naturally didn't show any ISDN service procedures being hit. After a week of cursing his god for not providing him with a bigger brain, he tried trace-pointing the lower level routines, got hits, and traced them back up to the rogue feature code.

A second good example of Layers is the overall M1 software architecture, described in detail in the next chapter, which goes something like this:



As an example of how this layering has helped us, the relatively narrow interface that the above "Intrinsics" layer provides between the SL-1 code and the hardware was very handy when we ported from Omega to Thor, or wanted to build emulators.

The M1 system is big enough and complex enough that most of the above layers can be further decomposed into their own layers.

3.1.3 Master-slave

Context: This architectural pattern, also described in *Pattern-Oriented Software Architecture*, is for partitioning work, preferably into semantically-identical sub-tasks.

Problem: Need to divide-and-conquer a piece of work to ease the load on the main processor.

Solution: The Master divides the work into sub-tasks, and delegates these to other processors. The slaves should not be aware that they are part of a larger structure; each should have whatever resources it needs to be able to operate independently.

Real examples: Patterns literature discusses a number of ways to use Master-Slave with redundant slaves to produce a more robust system. However, at present we really only use this pattern for load-sharing. The active Core CP is normally the master of the Network and Peripherals, who perform sub-tasks to handle calls. The roles are reversed in Symposium call processing. The off-board Call Center Server becomes the master, and the M1 CP is the slave.

This pattern is most powerful when a large number of slaves can be instantiated to do equivalent work in parallel. One such example in the M1 context is Network ACD, when a Meridian NAC drives a set of Meridian switches. At a more trivial level, the Core CP may ask a number of CTS circuits to play timed tones to a large number of listeners simultaneously.

3.1.4 Publisher-subscriber

Context: This pattern helps keep the state of multiple cooperating components synchronized. Like the previous pattern, Publisher-Subscriber is discussed at length in *Pattern-Oriented Software Architecture*. This design pattern is also known as "Observer", and may help you build a "Model-View-Controller" architecture.

Problem: Data changes in one place, but many other components depend on this data. When information changes, all of the dependent components must be informed of the change. You (always) want components to be loosely coupled, and you would prefer that the information provider not even know of the existence of the components that depend on it.

Solution: The Publisher-Subscriber paradigm separates the world into information producers and consumers. The name provides almost enough information for you to guess how to build it, although there are at least two basic implementations, which might be characterized as push vs. pull technology. As with the previous pattern, go look at the book for a really thorough discussion.

A weakness of this pattern is that it may become difficult to change your published interface once you have a number of entrenched subscribers (although the counter-argument is that not following the pattern doesn't make the changes any easier). This is in fact a general problem that shows up in all sufficiently-popular legacy systems. As mentioned earlier, even the proprietary interface to our digital telephones, which was never meant to be used by anything else, has been reverse-engineered and incorporated into a number of third-party systems which our customers now depend upon!

Real examples: Symposium provides a Publisher-Subscriber structure as a platform on which to build other services. For example, when the team wanted to make a web page showing how many calls had been handled, they just had to write a little client application that subscribed to the PresentCall event.

Thor's Hardware Infrastructure might also have been a good place to use this pattern (but it didn't...)

3.1.5 Corrective Audit

Context: You provide a service which hands out resources to clients. The clients are supposed to return those resources when they're finished.

Problem: Client code occasionally messes up. Items may fall off of the queues through improper usage, or because the code that was using them trapped. Syncronization may be lost between your state variables or resource objects and the hardware they represent (such as pairings between lamp and state or between call data and network connections). Remember that the M1 runs forever, so every lost resource is a big deal.

Solution: In the background, go look for lost sheep. State mismatches are fairly easy to detect. Items which have fallen off of a queue are a bit trickier, but the gist of that pattern is:

- Run through your resource pool marking all items as removable.
- Step through each known queue, including the free queue, marking each item as in-use. Don't allow interruptions within a single queue scan.
- Run back through the pool adding all removable items to your free queue. Possibly attempt to notify the previous owner that it's dropped something.

Corrective Audits are often designed as low-priority infinite loops. That is, they run in the background any time the switch has nothing better to do. Because this means that you can't simply block out other transactions and run to completion, your marking strategy must not get confused when other activities change the queues. In general, if you arrange your enqueuing code such that it marks each item as *in-use* whenever it adds it to any queue, the above algorithm will work. The first and last loops of the algorithm may be combined by careful coding. As we have discovered in the past, less careful coding is likely to kill valid calls.

Real examples: MAIN_AUDIT is the general auditor for system pool resources, checking the consistency of well over 100 conditions in data structures representing things like terminals, calls, network connections, and conference bridges. It normally runs in the background or as part of the daily routine schedule (also known as "midnight routine", see Overlay 17), but can also be manually invoked as Overlay 44.

LAMP_AUDIT ostensibly exists to make sure key-set lamps correctly reflect the current call state. It normally audits one lamp/state pair per timeslice. However, people have taken advantage of the fact that it was a convenient periodic task to do all sorts of other stuff, such as the Low-Speed Link audit, the attendant console Busy Lamp Field audit, and Idle Trunk counting.

The overlay supervisor keeps private lists of the TNs, DNs, etc., that each craftsperson is accessing to ensure mutual exclusion, and these lists are audited by routines like SBA_MU_AUDIT.

3.1.6 Memory manager

Context: Many independent applications need to allocate memory dynamically.

Problem: The M1 system is scalable and configurable, so you don't know at compile time how much memory you'll need for each application. On the other hand, using the heap consumes real-time, is susceptible to memory fragmentation and leakage, has unpredictable failure modes, is hard to audit, and makes debugging trickier.

Solution: Create a Memory Manager. The salient characteristics to remember when designing such a system are:

- Allocate memory infrequently (ideally only on restarts), in large contiguous chunks, and split these into blocks, avoiding memory fragmentation.
- Structure the blocks into queues, including a free block queue. A dequeue operation can be much faster than a generalized *`malloc`*, which has to search for a block of the right size.
- Associate the blocks with an owner, to help trace memory leaks.
- Audit the blocks to make sure they're all on queues. Ideally, the audit should pass the blocks to an application-provided routine to confirm that state data is sound.
- Provide standard tools to track memory usage, and to print out block contents.

Real examples: The big SL-1 Memory Manager is the Call Register system. While CRs were initially created to track call states, they have been reused as a generic managed memory system. Among its Corrective Audits, `MAIN_AUDIT` invokes `CRAUDIT` to ensure call register queue sanity.

Notice that this `CRAUDIT` is really a garbage collector. Despite my earlier claim that we can't afford garbage collection, we can afford `CRAUDIT` because of two simplifying properties of the system. The first is that all CRs are the same size, so recovering a block doesn't require any rearrangement of memory. The second is that the chore of picking up CR garbage is the exception rather than the rule. Regular call processing politely returns all CRs to the queue. `CRAUDIT` should only expect to find calls that died violently. Therefore, the workload is pretty light, and the number of disconnected CRs that `CRAUDIT` hasn't yet found will always be too small to affect performance.

The rest of Thor (the non-SL-1 part) now uses Seaweed, which replaces the standard `malloc` and `free` procedures. Seaweed follows the Memory Manager pattern.

3.1.7 Safe interpreter 👍

Context: Non-experts want to write features too.

Problem: It's too expensive for us to write a feature that only one customer needs. That doesn't stop the customer from needing it. But the system itself, along with the supporting toolset, represent a prohibitive learning curve, even if we were inclined to allow customers to tamper with the code.

Solution: Provide a "programming environment" which is simple enough and safe enough that customers can write their own features. Do this by providing a virtual machine, with a controllable interface to the real box, and some extra abstractions to make coding easier.

As a sweeping generalization, interpreters enable easier development environments than compilers. If nothing else, you can try things out instantly, without having to wait for a compiler to run, but high-level debugging tools can also usually leverage the interpreter's environment to allow arbitrary examination of variables, single-stepping through lines of source code. Because the abstractions the customer needs are high-ish level building blocks, the size of the customer-written code is likely to be small-ish, so you can probably afford to interpret it, so do. Besides, as Gosling points out in his defense of Java byte-code, an efficient interpreter may theoretically be *faster* than native machine code on any processor whose power is limited by memory fetch speed (like most of ours).

A Safe Interpreter effectively gives you a big leap in flexibility and time-to-market.

Real examples: CCR scripts allow M1 call center customers to script their own call flows.

The other obvious example from the computing industry is Java (or maybe it's called Java** by the time you read this). There is at least some talk of putting a Java Virtual Machine onto the M1.

Extremely real example: The archetypal Safe Interpreter was my driver during a journey along the Afghan border. He spoke six languages, could fix the truck, and knew his way around the local tribal customs, providing a safe interface between his customer and a pretty gnarly reality. He very effectively converted naïve requests into reasonable actions, without allowing us to stray into the (many) nearby areas that would prove unsafe.

3.2 Not-so-good patterns

3.2.1 Busy wait

This pattern has many flavors, but is often a misguided version of the Transaction Engine. You usually have a choice between designing your lowest layer around interrupts or polling. Interrupts stop you in the middle of what you're doing. Polling requires you to go and check if anything needs doing. In a hard real-time environment, interrupts are more usual, but both have their place. Once you get above the lowest layer, you really want the software to wait quietly for work to be fed to it. Any other action ought to be unnecessary.

A Busy Wait eats up CPU cycles to no visible advantage. It says, "Are we there yet? Are we there yet? Are we there yet? ..." until there's something to do.

In general, a priority-based system can tolerate at most one Busy Wait with no impact on performance, and it will only run when there is really *nothing* else to do. In contrast, a class-based scheduler (which we VxWorks doesn't directly provide, but WORKSHED sort of is) allows Corrective Audits, for instance, to be written as simple infinite loops and then manages to allocate them only a portion of the CPU time.

The most common place to find Busy Waits in M1 code is when a task needs a very short delay. These Busy Waits should be avoided, as they not only waste time, but tend to spawn errors whenever we speed up the CPU. If you *must* do one, the following unusually portable copy is provided for your cloning pleasure. It came from the set download code, buried deep inside TCM_OUTPUT_MSG:

```
START_TIME := RTCLOCK;
WHILE ^OUTPUT(OUTPTR) DO
   IF RTCLOCK - START_TIME >= .DL_WAIT_TIME THEN
   BEGIN
      BUFFER_OVERFLOW(.ERR30,570,OUT_DATA[2]);
      RETURN;
   END
```

The SL-1 "tier" processing within WORKSHED also uses a Busy Wait to poll for input, monitor Ring Again calls, etc.

3.2.2 Global variables

This is a tactical idiom for doing lots of different things. Global (or at least shared) buffers can be very effective at improving real-time performance. But Global Variables can also be used to avoid careful design; as long as all important state information is kept in Global Variables, you don't have to think very hard about what your API should be. The entire POOL module is an extended example of the Global Variables pattern.

Problems with the Transaction Engine pattern frequently stem from designers' having made the Progress Mark a Global Variable. To begin with, anybody can change its value. Designers sometimes take advantage of this to short-circuit the normal state transition progression. This works, but can make it very difficult to understand the underlying logic of the state machine. The other big problem appears when we need to run more than one copy of the state machine (for instance, when two attendant consoles are operating on the same M1 switch). What we need are private instances of the Progress Marks, but this has not generally been done, which means that the code generally breaks. In the case of consoles, the procedure `SETATTNPTRS()` was created to make sure that the global variables are refering to the right console instance, but we end up having to invoke it billions of times per call.

Two examples in one

The overlay input processor is a reasonably useful piece of SL-1 code. However, it actually contains both the Busy Wait and the Global Variables patterns. Client code gets invoked on every timeslice, and has to check `SCINPUTMSG` to see if there is any reason for it to have been called. If it only checks `SCINPUT` to see what the input is, it will produce random behavior. Both of these variables are global, making it difficult to handle multiple terminals. `SET_SLICE_VARS` sets up all the right globals to get around this, but what you really wanted was for each terminal to have private copies of the variables.

3.2.3 Replicated data 👎

The M1 is a network, and it is frequently efficient to keep copies of some information at more than one node of that network. For instance MAT and the M1 Core each have a replica of most of the configuration data. One example of how unwieldy this can get is the number of telephone directory databases an enterprise ends up maintaining. Ever try to get your phone number changed in "all" of the Nortel databases?

Especially within a node, the temptation to replicate data should be resisted. It's far easier to keep data correct if there is only a single master version, and all other applications that need the information query the master for it. (If everyone gets a pointer to the data, references are faster, but it gets harder to make changes. Consider the case when data is deleted but some user continues to reference it. It kind of works for a while, but probably has stale data, until that portion of memory is reallocated and written over by someone else. These are really tough to track down, because the appearance of the symptom is linked to some unrelated event!) If real-time or survivability concerns make replication unavoidable, then we must at least think through a protocol for ensuring that the copies can't get out of sync, and we probably want a Corrective Audit too.

3.2.4 Overlaid data 👎

In the embedded systems philosophy, bits are precious. The first SL-1 had only 16K words of store in total, including program store, protected and unprotected data, and I/O buffer space. People try to save bits, especially in structures that have many instances like Call Registers, by using the same field to mean several different things in different contexts. This was so important to our early designers that the SL-1 language makes you manually code the word in which you want each element of a structure to appear.

The technique is reliable *if and only if* the different uses of a given bit can't collide. The problem is that, as features evolve, features that were once mutually exclusive start to cohabitate. If one feature overwrites data stored by second feature, the latter will behave randomly. Sometimes, the random behavior will even be correct, accidentally. Like all intermittent problems, these bugs are very tough to understand, even if they are easy to fix once understood.

3.3 The usual suspects: Patterns of problems

Given the claim that many characteristics are ubiquitous in the M1 design, it should not be surprising to see that certain villains are repeat offenders. Apart from the issues already highlighted in the above Patterns discussion, expect the following kinds of failures.

3.3.1 Race conditions ☠

Race conditions are the first of a broad class of ills that are endemic to multi-process designs. Race conditions happen when two independently-running processes modify information about a single real entity, resulting in non-deterministic behavior. In other words, if process A gets there first, the results are different than if process B does. Within the M1 Call Processor, many independent processes send messages to each other. Also, as discussed in the previous chapter, the M1 itself has become a network of components which communicate through messaging. A further layer out, the M1 connects to the PSTN through trunks using one or more different messaging standards. All this asynchronous messaging leads to many different strains of race condition.

The worst problems occur when a task thinks it knows about a state that has really changed, and continues about its work without checking. The farther it gets, the more likely it is to be trampling something, and the more backtracking will be necessary to get back to a sane position.

Race conditions are nasty problems to debug, because they tend to happen intermittently. One class of "Heisenbugs"[14] traces to race conditions. Other race conditions happen only during the customer's busiest hour. Particularly malevolent ones happen only during demos. The intermittence of race conditions means that the best way around them is by careful analysis and planning, rather than by hoping to detect them during test.

One way to help software to detect race conditions (and other maladies) is to design handshakes into protocols. The M1 has had problems with oversimplified download protocols, where the absence of an acknowledgment to a message might mean:

(a) I didn't receive your message,
(b) I got it, but I have no idea what you're talking about, or
(c) I got your message, I understood it, and I'm doing just fine, thanks.

The Transaction Engine pattern also helps avoid some race conditions, but they still happen.

3.3.2 Deadly embraces ☠

This is the friendlier European name for "deadlock". Like race conditions, it is a problem stemming from having multiple processes running, but this time each is waiting for the other to say something, or to release a semaphore. Another cause of deadlock is Busy Waiting. If a high-priority process is in a tight loop checking its input buffer, it may be preventing another process from being able to send messages to that buffer.

Peer-to-peer messaging, where neither end has priority over the other, is especially vulnerable to deadlock if the protocols aren't designed carefully, especially if both ends have "blocking" semantics, that is they wait for a response

[14] Errors which change or disappear when debugging tools are turned on (after Werner Heisenberg's work in quantum physics).

to each message. The usual safety precaution is to run a timer which wakes you up if you don't hear from the other guy.

In contrast, "livelock" is a situation in which some critical stage of a task is unable to finish because its clients perpetually create more work for it to do after they have been serviced but before it can clear its queue. Livelock differs from deadlock in that the process is not blocked or waiting for anything, but has a virtually infinite amount of work to do and can never catch up. This doesn't have to be an overload situation; it could just be a bug that, in the process of handling a message, recreates the same type of message.

3.3.3 Constipation ☠

This is yet another pattern common to miscommunicating processes. A process sending a lot of messages may run into this problem, which in turn can lead to deadlocks. The problem is that all the buffers get filled up because nobody is reading anything. This is especially likely if the reads are being done at a lower priority than writes. One reason to do some flow control at the leaf nodes it to reduce the risk of constipation further along the tree.

A constipated process will either hang, or start dropping messages on the floor. Either case will cause you grief, but if you can't prevent the situation, you must at least choose which of these two failure modes is likely to be least damaging.

The classical place to discover Busy Waits is in output code that doesn't want to cause constipation, on a system that doesn't provide a good mechanism for very short delays. For instance, when we build faster CPUs, we often discover that `LAMP_AUDIT` starts sending out messages faster than the rest of the equipment can handle. We should probably expose VxWorks' `taskDelay()` or even `nanosleep()` to SL-1 to remove this excuse...

3.3.4 Software version mismatches ☠

A classic recurring networked-computing hurdle is handling software upgrades. You can't change the software on both ends of a communications pipe at precisely the same instant. This leaves you with one of the following choices:

- accept random results during the phase in which the versions are mismatched
- take (at least a portion of) the system out of service while you upgrade all components

- pay a performance price to make your protocol completely version-independent
- freeze the interface
- pay a code-bulk price to have version-tolerant designs

None of these choices is really ideal, but usually at least one is tolerable in any given situation. The crises come when the problem was forgotten, and the choice not made explicit, or when a collection of components carefully code up different choices.

3.3.5 Partial failures ☠

M1 is a network, and most of the network must be working for anything useful to result. One thing that happens to us sometimes is that a portion of the system gets in trouble, and reinitializes. If other nodes don't hear about this, the Replicated Data they have about this node's state will now be incorrect. From this point on, any number of symptoms may appear. This is one of the reasons for Corrective Audits and handshakes.

3.3.6 Too many tasks ☠

The root cause of any of the above problems may be a design based on too many asynchronous tasks. The OO design paradigm asks you to think about all your objects as independent communicating entities. It's a useful discipline, and may help with component reuse or object distribution across platforms. However, just because you've formulated a *design* this way doesn't mean you always have to *code* it this way. Your elegant diagrams may result in slow code with severe non-deterministic wobbliness. Sometimes all you really needed was a procedure call.

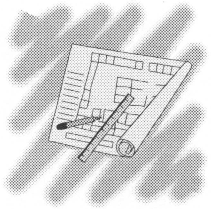
3.3.7 Power and grounding problems ☠

Meridian 1 seems to be plagued by these, especially in Florida. Repeated hardware problems can usually be traced to this source, although on the older systems faulty cabling also gave us many headaches. Pournelle's Law[15] holds that if you ever have any computer problem, check your cables. Successive recent generations of our PBX have tried to reduce our problems by simplifying the cabling—probably quite a sound plan of attack.

[15] Jerry Pournelle, science fiction writer and senior contributing editor to Byte magazine.

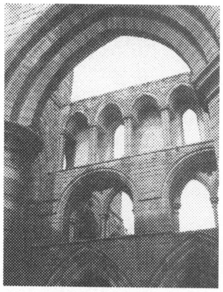
Part III

Software Architecture



4. Core software architecture

Αρχιτεκτων: The word and the sensibility are ancient Greek. "Architecture" evokes designs that mix clever engineering with fine art, a coherency of structure and ornament, a delicate balance of forces. One Victorian gentleman lectured, with charming bombast, "No person who is not a great sculptor or painter *can* be an *architect.* If he is not a sculptor or a painter, he can only be a *builder.*"[16]



Alan Kay pointed out that the architecture of gothic cathedrals came from people with the art and science to imagine fantastic stone structures that were mostly air; the pyramid builders, while also impressive in their own way, basically made rock piles. The cornerstones of a software architecture are the long-term decisions we make about where we allow or restrict flexibility in our structures. Kay is one of a number of luminaries who have lately argued to varying extents that the correct highest-level software architecture is necessary, and perhaps nearly sufficient, to produce all manner of good things in the final product: efficiency, correctness, robustness, reusability, even beauty. In essence, the argument is yet another return to Dijkstra's separation of concerns.

Part III of this book describes our core software structure. If the M1 is not quite a cathedral, neither is it fair to say that it was built with no thought of architecture. The problem is more that the architecture was best suited to the original specifications of the building. It has since undergone many renovations and extensions by different owners, and has a new foundation, new siding, and new plumbing. If not pristine, it is at least full of character and more or less functional, with only a little dry rot.

This chapter will attempt to cover the big picture, using the descriptive models Kruchten espoused in his "4+1" methodology.[17] These amount to five different

[16] John Ruskin, whose works included *The Poetry of Architecture* and most of the 19th century's art criticism.

[17] Phillipe Kruchten, IEEE Software, November '95; see http://www.rational.com/support/techpapers/ieee/

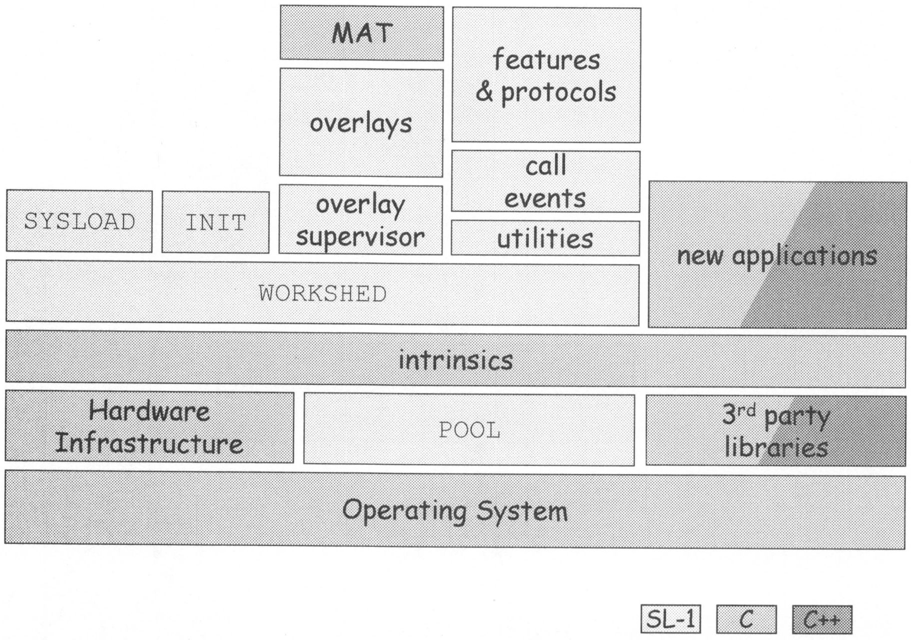
views of the same architecture, which between them capture all of the major aspects of the design. The remaining chapters of the book delve into a bit more detail—not everything you'll ever need to know, but hopefully enough to get you started. With teams on three continents busily modifying the software, you'll probably need to go look at the code if you want to understand the next level of detail with any reliability.

4.1 The logical view

At the top of the software layers are the two basic interfaces the M1 has with the world. On the left is the craftsperson interface (MAT and the overlays); on the right is call processing. SYSLOAD and INIT are used to reboot and restart the system, respectively. WORKSHED is the scheduler for all SL-1 software. The Hardware Infrastructure (HI) is the maintenance framework built by the Thor team. POOL is the single module that declares all symbols (data types and procedure names) shared between SL-1 modules. The third-party things include memory management, protocol stacks, and assorted spare parts. And the operating system is currently VxWorks, although we have made several additions.

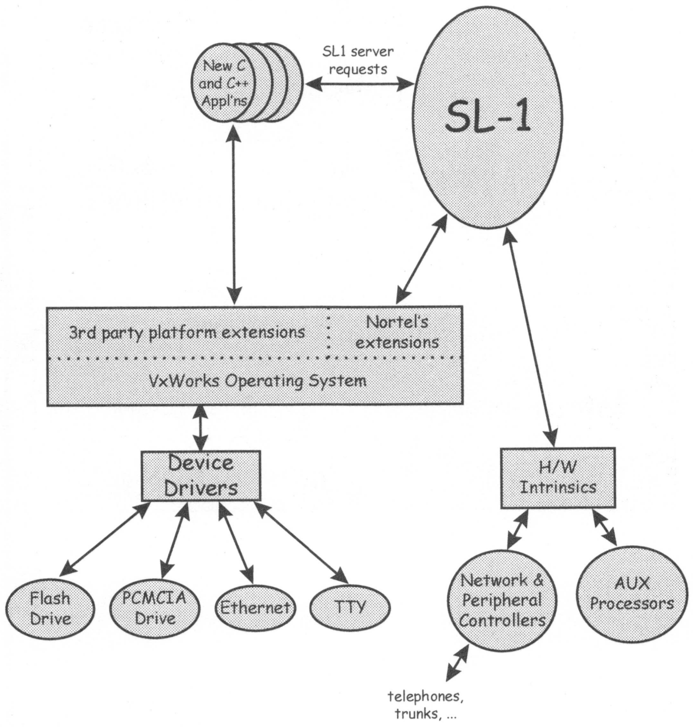
The current structure reflects the evolution of the product. In the beginning, the SL-1 software used to do more or less the whole job of running the switch, so it needed its own scheduling and management functions. SL-1 code communicates with the real equipment through a family of procedures called intrinsics. Because this interface is fairly narrow, several generations of porting from one processor family to another have been relatively painless. In 1990, one of these porting projects, Thor, encapsulated the SL-1 software, and hoisted it on top of a commercial real-time operating system, VxWorks. VxWorks in turn now provides a platform for many new services, typically coded in C or C++. Among these new applications, the biggest are MAT, Mobility, and SCCS.

Each of these components will be discussed in more detail in the coming chapters.



4.2 The process view

The run-time relationships now look roughly like this:



Maybe the first thing to notice here is the prominent blob labeled "SL-1". As stated, there was a time when almost all of the PBX software was written in SL-1, and it is still responsible for the bulk of the traditional PBX-type work. It has exclusive responsibility for the call processing state machine. It runs all of the calls, manipulates most of the configuration data, keeps the call processing hardware in service, and audits everything to make sure it's all still okay.

Until recently, if the SL-1 task trapped, it was quietly restarted. Other calls, and certainly other tasks, were unaffected. This gave us some troubles synchronizing states between SL-1 and non-SL-1 code, so now we restart the whole machine. The original model was probably the right idea, though, and we are likely to be pressed back in this direction to improve reliability of non-SL-1 applications and decrease recovery time.[18]

The other key observation is that the operating system is not a "process" at all.

4.2.1 How processes talk to each other

The different pieces of software communicate with each other through a variety of mechanisms:

- For intra-SL-1 message passing, shared memory is probably the most common. It's fast, which still matters. But it's susceptible to corruption if more than one process is writing to the same database, which is why we try to keep all call processing state data isolated to SL-1 code. There are many implicit conventions about how SL-1 global variables will be used. Procedures control subsequent actions as side effects of updating these variables.
- Asynchronous inter-process communication can be with VxWorks messages and semaphores, but also sometimes with shared memory, which sometimes works.
- Direct procedure calls are the norm with the core processor, as they are well optimized for speed, and avoid some scheduling complexities.
- Recently-added software has made some use of Remote Procedure Calls (RPCs) and Common Object Request Broker Architecture (CORBA).

[18] What we really need is the concept of "call death", as opposed to "process death". Each CR is sort of a lightweight task control block, and usually the other CRs have valid data. Don't kill SL-1; kill this call.

4.3 The physical view

The physical layout of software is determined both by history and by engineering considerations. It has a great effect on the "non-functional" aspects of operation that were discussed at length in Chapter 2.

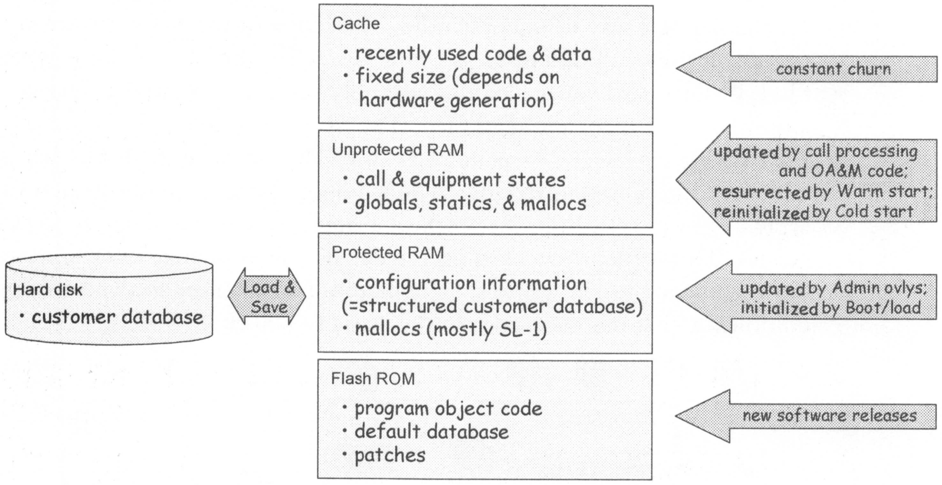
4.3.1 Mapping jobs to processors

On the original SL-1 PBX, this was an easy decision. You had one processor. It did everything.

The core CPU got overburdened, and the code got more complicated. The software was poorly documented and brittle. It became increasingly hard to make substantial changes to the SL-1 code without breaking something. Delivery of SL-1 code changes was locked to global software release schedules. Features written in SL-1 could obviously only be sold to M1 customers. And in any case, you couldn't hire designers who understood the SL-1 language. Pretty soon, the decision began to look equally easy, although the answer was different: do things anywhere but the main processor. This led to off-board processing for ISDN, microcellular, OA&M, voicemail, various ACD components, and CDR formatting.

Then a curious thing happened. Some designers began to realize that SL-1 was actually a reasonable vehicle for call processing. There is an opinion (by no means universally-held, but the faction is growing) that any code which is contingent on call processing states should be done not just in the core processor, but in SL-1. Signal processing, on the other hand, is well suited to off-board processing, since after all the PBX is expressly designed to stream voice data from one place to another. CDR formatting and GUIs are also sensible things to run elsewhere, but care must be taken to engineer both the size of the pipes going to the off-board processors and the priority of the tasks servicing those pipes.

4.3.2 Mapping data type to store type


Cache
· recently used code & data
· fixed size (depends on hardware generation)
constant churn
Unprotected RAM
· call & equipment states
· globals, statics, & mallocs
updated by call processing and OA&M code; resurrected by Warm start; reinitialized by Cold start
Hard disk
· customer database
Load & Save
Protected RAM
· configuration information (=structured customer database)
· mallocs (mostly SL-1)
updated by Admin ovlys; initialized by Boot/load
Flash ROM
· program object code
· default database
· patches
new software releases

This whole issue is covered in detail in the Platform chapter, but there are a few overriding guidelines.

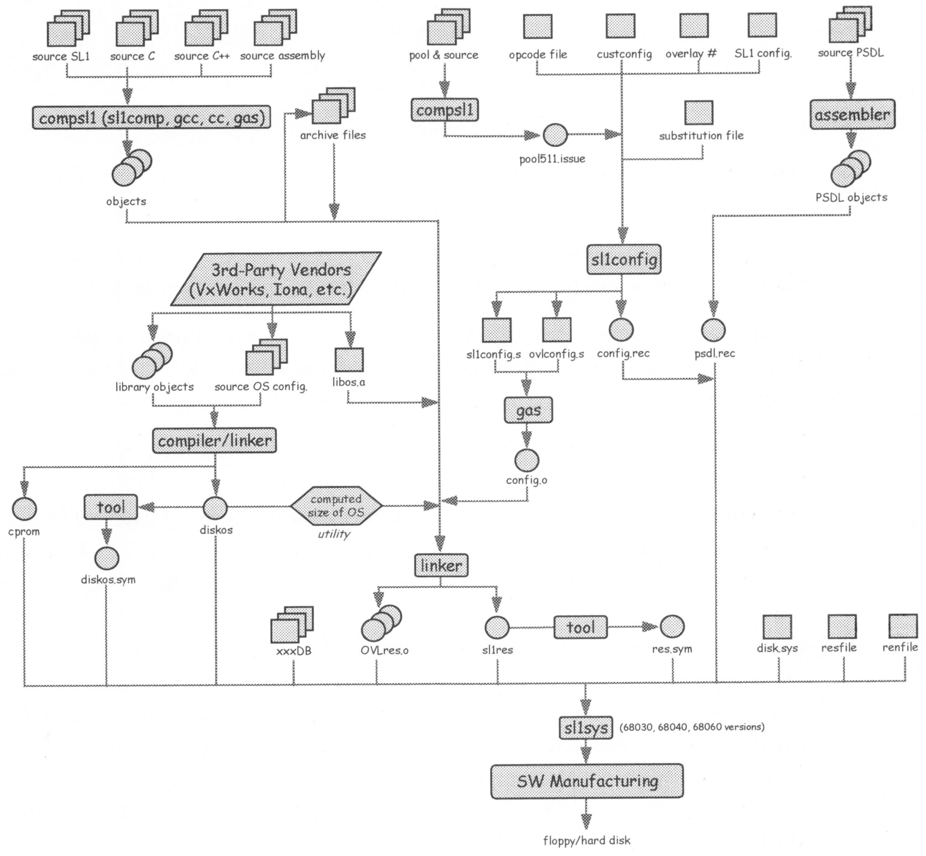
Caching isn't planned, it just happens. Whatever was most recently needed is in there. This is sometimes the same as what will soon be needed, but often not.

Unprotected Data is for things like state data that changes frequently. Protected Data is for more permanent stuff. The time to read both types is about the same, although writes to protected data are noticeably slower.

The bulk of feature and protocol logic goes into the Call Processor, historically in the SL-1 code, and now increasingly in C or C++. ISDN chose not to follow this rule, instead putting a lot of logic into the peripheral card. This helps performance if the messaging overhead to the core CPU is small, and it speeds development if the interface is clean and simple. However, they must resolve many of the standard Ugly Patterns in distributed computing. Also, ironically, they will have to pay a substantial development price to port the functionality if (as is expected) the next generation of M1 doesn't have standard Network Equipment. Normally, putting the logic onto an adjunct processing board should mean close to zero rework if we change the main architecture. Despite the problems with SL-1 coding, on balance life is probably easier if call processing code is implemented there and peripheral software doesn't try to understand about call states.

4.4 The development view

Maybe the best way of understanding the scope of the M1 software challenge is to look at the loadbuild process. The source for the M1 system is written in SL-1, C, C++, and various native assembly languages, as well as taking input from a number of control and data files. The SL-1, assembler, and C source lives in the PLS library. The C++ is in ClearCase. The source code for Orbix is in hiding, and was last spotted in Ireland. The loadbuild process draws from a vast-ish array of these source files to produce a single executable image for a particular switch. Don't focus too hard on the details. They change over time, and between development groups. The picture is really here to demonstrate the complexity that has evolved into the static structure.

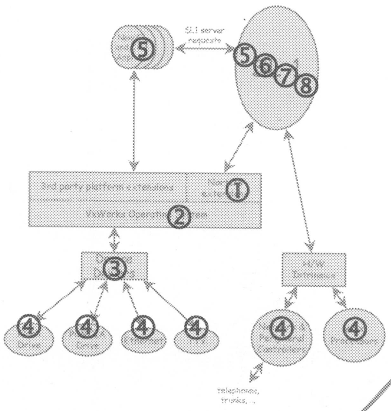


4.5 Scenarios

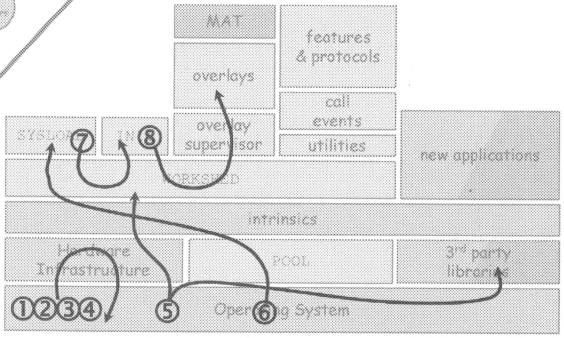
4.5.1 Cold restart

If a user presses the MANUAL RESET button on the CP card, the system will be rebooted. That is, the database will be reloaded (usually from disk) and all of the key data structures will be reinitialized. Unlike a warm restart, the active calls will not be resurrected.

Looking at the steps in detail, the first thing that happens in this scenario is a hardware interrupt①. This signals VxWorks to restart the operating system②. This in turn restarts the various device drivers③ enabling the hardware④, and recreates all of the main tasks⑤, including `tSL1`. When `tSL1` is recreated, `WORKSHED` is restarted and it in turn invokes `SYSLOAD`⑥ to reload the customer database. Once `SYSLOAD` has finished, `WORKSHED` asks `INIT`⑦ to reinitialize all of the major SL-1 data structures and polls the network hardware. Finally, `WORKSHED` launches various background activities⑧ and we're ready to handle calls.



Process View

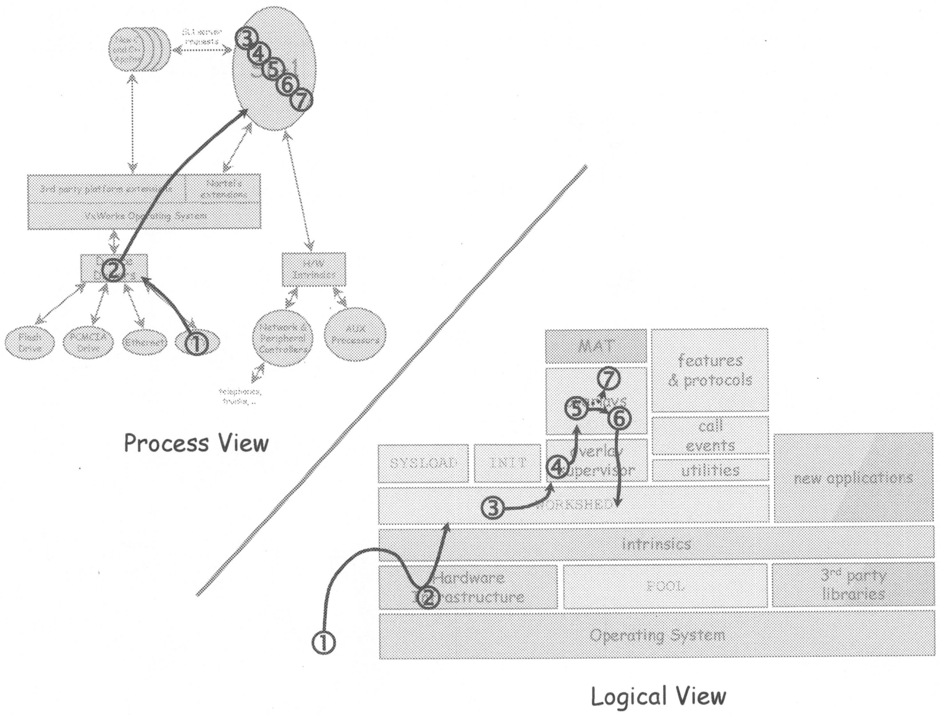


Logical View

4.5.2 Service change

A craftsperson makes changes to the customer database by typing commands at a TTY terminal.

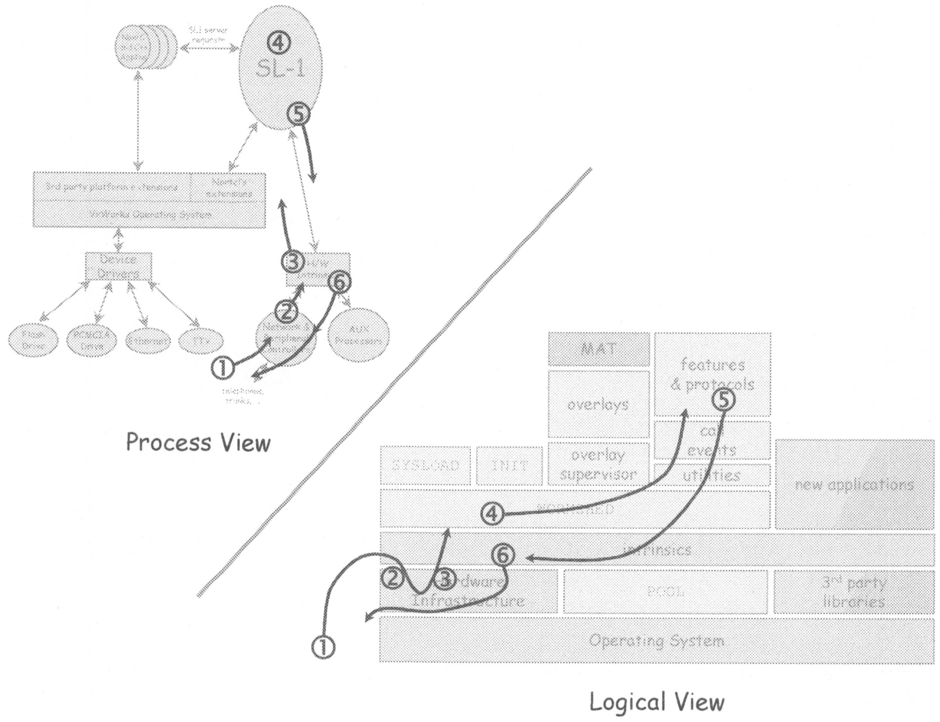
Looking at the steps in detail, each keypress generates an I/O interrupt① into the CP. The TTY interrupt service routine queues the character in the TTY input buffer②. `WORKSHED`③, in one of its infinite loops, notices the character and passes it up to the overlay supervisor④. This in turn passes it up to the service change code⑤ for whatever overlay this TTY has active. The overlay will attempt to parse the command, and will usually discover that it needs more characters, and just go back to sleep until they arrive⑥. This process will repeat itself, building up the command character by character. Finally, once the whole command has been typed, the overlay will do something useful⑦.

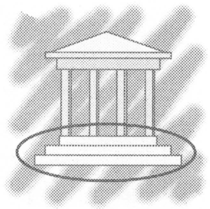


4.5.3 Telephone call

A phone call is composed of a series of transactions, each of which goes roughly like this.

The telephone set (say a digital set) sends a TCM signal① to the peripheral. This gets converted into the older SSD format②, and relayed on to the CP via the hardware intrinsics. The network input interrupt causes the ISR to fetch the message from the Peripheral Signaling card on the NE shelf, and enqueue it for call processing③. `WORKSHED` sees the message, finds the associated terminal, and reacts to it based on the current terminal and call states④. It may also send a message back through the same path⑤ to either the originating or terminating terminal⑥.





5. 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.).

5.1 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.

5.2 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[19] 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).

5.2.1 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).

[19] 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.

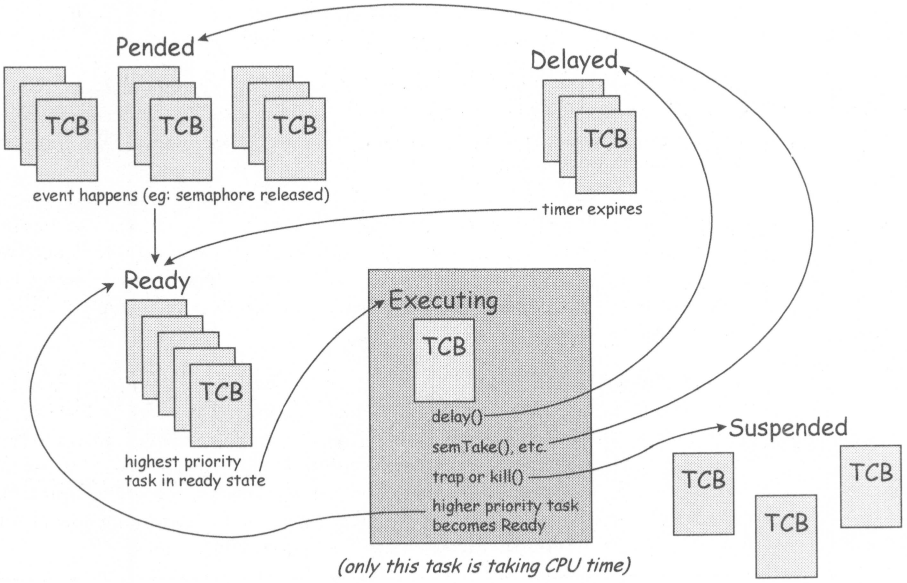
Bonus advice about stacks: Never return a reference to a procedure's local variables. Locals are kept on the stack, which gets reused by other procedures after you return. The simplest case actually works: in the absence of interrupts, the procedure which just called you can see anything you left on 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 `malloc`s 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.

5.2.1.1 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.


Pended
TCB
TCB
TCB
event happens (eg: semaphore released)
Delayed
TCB
timer expires
Ready
TCB
highest priority task in ready state
Executing
TCB
delay()
semTake(), etc.
trap or kill()
higher priority task becomes Ready
(only this task is taking CPU time)
Suspended
TCB
TCB
TCB

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…).

5.2.1.2 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?

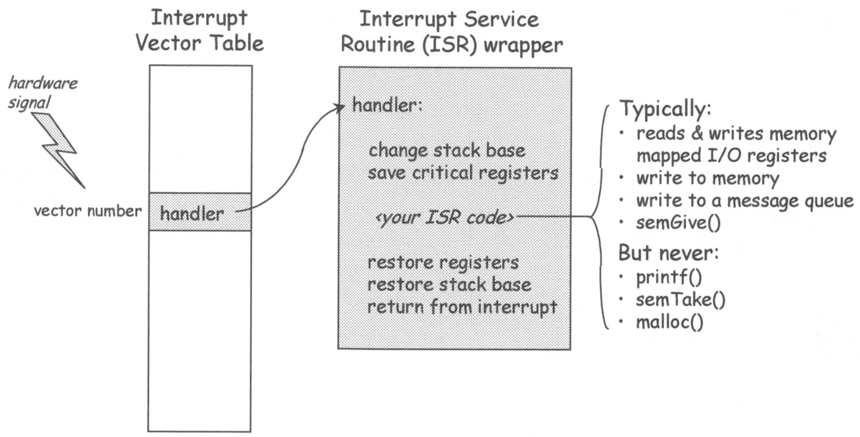
If the most important job of the PBX is call processing, you might expect `tSL1` to be much nearer to the top of the list. The problem is with `WORKSHED`, the SL-1 work scheduler that acts as a mini-OS task manager. It controls a number of independent activities, some of which are audits that run whenever 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.

5.2.2 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.

5.2.3 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.)

5.2.4 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.

5.2.5 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?

5.2.5.1 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

5.2.6 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.

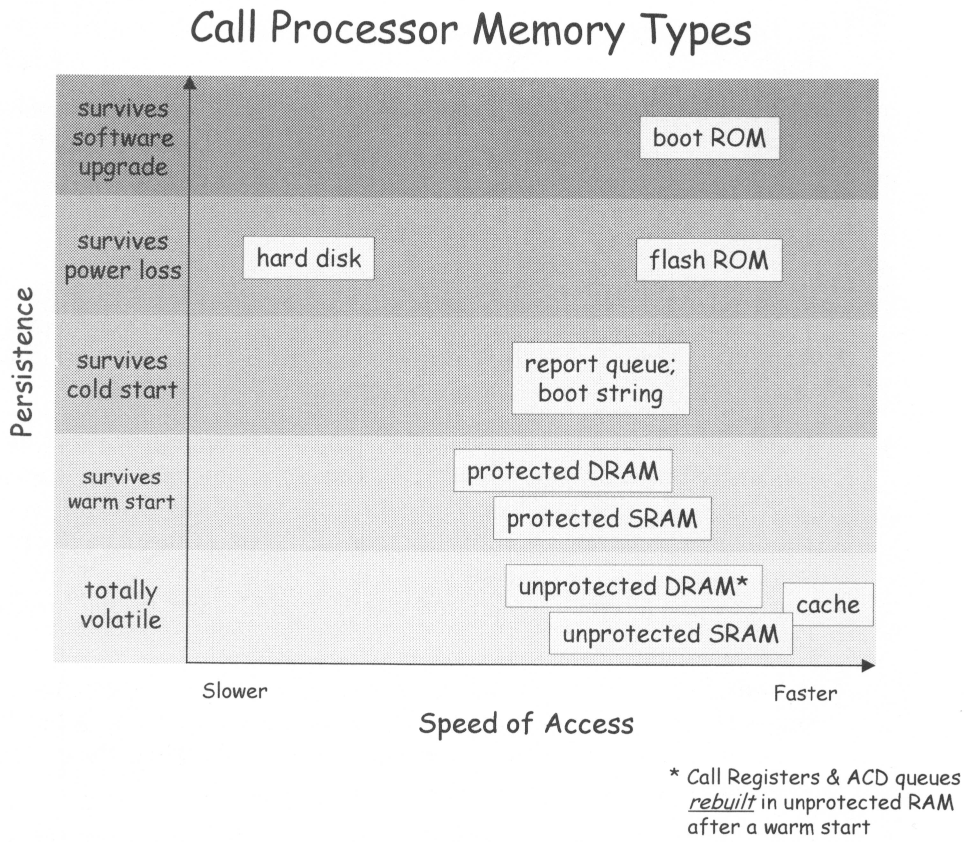
- The Standard I/O Library contains macros as well as routines. As usual, breakpoints can't be set in macros.

- Mobility, SMP, ISDN, QSIG, MMIH, and ICCM all now use the OS heap.

5.3 High availability: Our modifications to VxWorks

5.3.1 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.

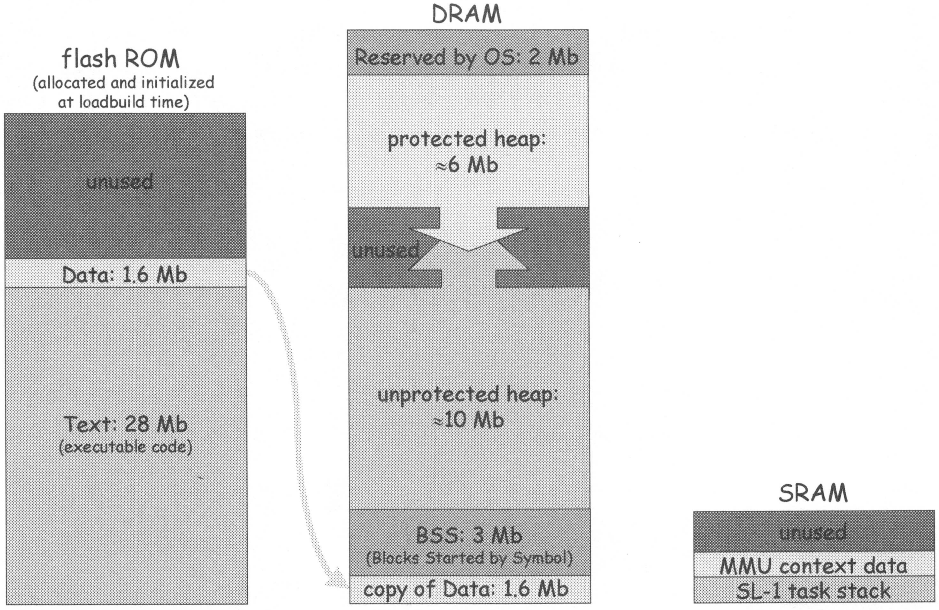
A further level of reliability, already discussed in Chapter 2, is provided by using the MMU to make certain areas of RAM **protected** against inadvertent corruption. We require people to explicitly unprotect data, change their variable, and then reprotect it. Of course, during this window, they are 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.[20]

[20] 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.

5.3.1.1 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.

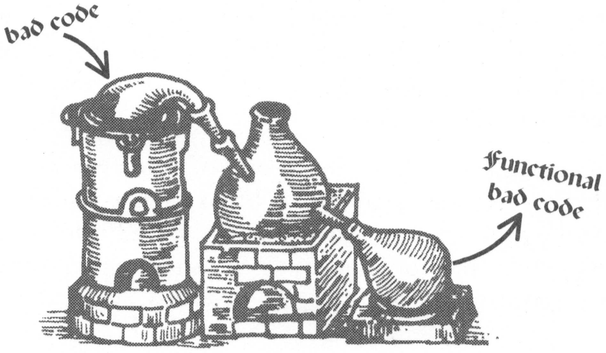
5.3.1.2 Conjuring with bad pointers

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/).

5.3.1.3 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

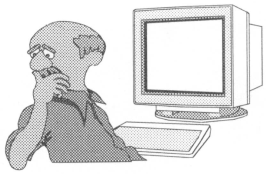
2) the OS shall be "just a collection of libraries" that an application links to as if they were any other libraries.

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.

5.3.2 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*.

5.3.3 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.

5.3.4 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.

5.3.4.1 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.

5.3.4.2 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.

5.3.4.3 Cold restart

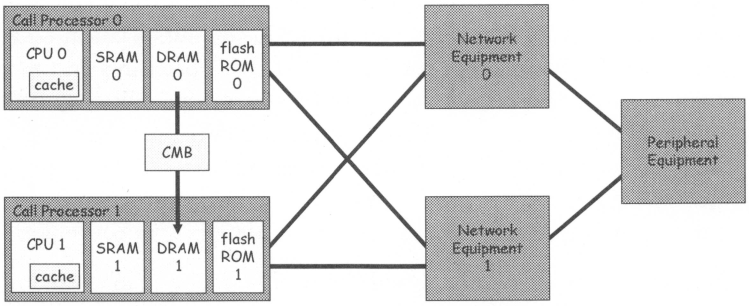
This is part of our normal software install sequence, although it may also happen if the hardware watchdog expires, the CP card is reseated, or its manual reload button is pressed. `SYSLOAD` reloads the database (and on non-flash machines, the code too), usually from a hard disk. Unsaved database changes 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.

5.3.5 Dual CPUs

5.3.5.1 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.

5.3.5.2 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 ¼ second, 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[21] 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[22] 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.

5.3.5.3 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.

5.3.5.4 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.

5.3.5.5 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

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

22 At least Chorus, Tandem, and Sun, and probably others before we're done…

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.

5.3.6 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.

5.3.7 Tools

5.3.7.1 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.

5.3.7.2 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.

5.3.7.3 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.

5.4 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.

5.5 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.

5.5.1 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.

5.5.2 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`.

5.6 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.

5.7 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.

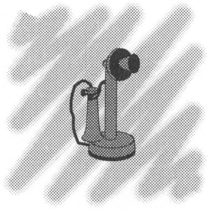
We don't yet have any way of doing symmetric load-sharing between multiple CPUs, although we are discussing some possibilities.

5.8 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[23]. 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 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.

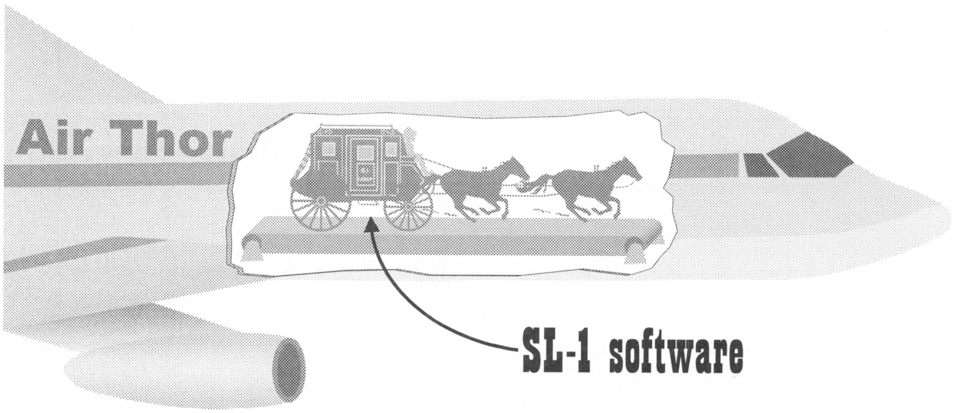
[23] According to the hacker jargon file, this excellent term derives from Swift's "Gulliver's Travels" via the Danny Cohen's famous paper "On Holy Wars and a Plea for Peace" [USC/ISI IEN 137, April 1, 1980]. The Lilliputians, being very small, had correspondingly small political problems. The Big-Endian 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!



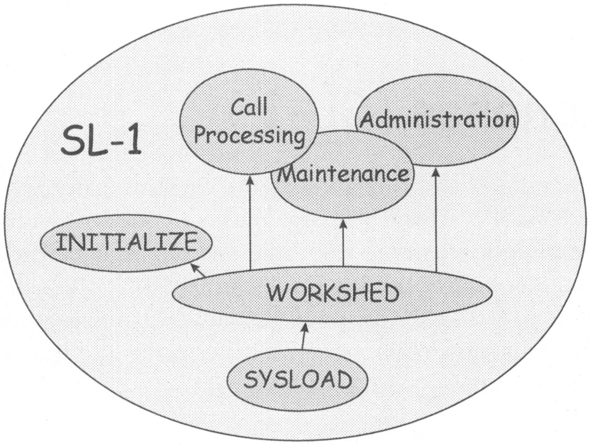
6. The phones (SL-1)

The overall M1 software architecture diagram featured a prominent bubble labeled "SL-1". This is the large mass of code written in "Switching Language 1" whose evolution dates to the earliest of our digital switches. Prior to the Thor project, that bubble used to do the entire PBX job, including stuff like scheduling, I/O, memory management, and initialization. In fact, SL-1 probably still thinks it's in charge, since we never really told it that it's now just another application running on top of VxWorks.

To over-dramatize the picture:



It shouldn't be too surprising that the SL-1 code in turn can be decomposed into a structure very similar to that of the M1 system as a whole. Roughly speaking, we have the following run-time relationships:



The SL-1 the code still assumes nearly all responsibility for managing telephones and calls.

- `SYSLOAD` (global procedure #0) is invoked by the firmware on a reboot. Its primary responsibilities are to load the customer configuration database into protected memory and enable all TTYs. Then it cranks up `WORKSHED` to manage the distribution of work.

- `WORKSHED` is the work scheduler. VxWorks treats all of SL-1 as a single task, `tSL1`. Within this task, `WORKSHED` ensures everyone gets a chance to run. `WORKSHED` starts by calling `INITIALIZE` to set things up, and then loops forever (barring catastrophe), sharing work between the remaining bubbles. Each of these manipulate *the same* interdependent set of call and terminal data, so it is critical that `WORKSHED` coordinates their activities in a way that keeps them from trampling each other's work. (See the Transaction Engine pattern in Chapter 3.)

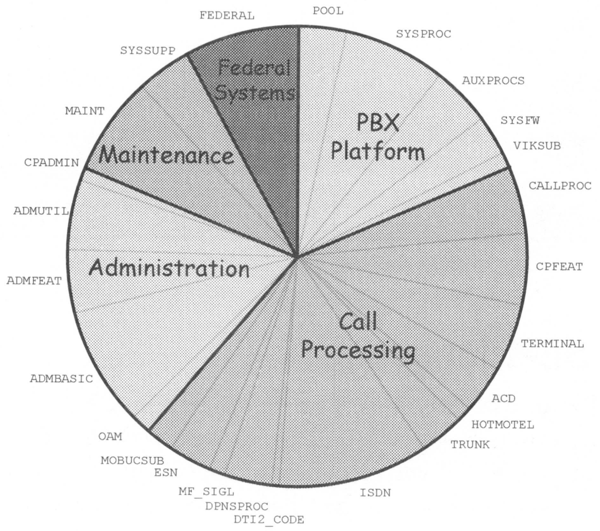
- `INITIALIZE` initializes the call related data structures in the system, and the I/O system, and then returns control to `WORKSHED`.

- Call Processing code receives incoming messages from the various terminal types, does digit translation, connects speech paths, and generally runs the call state machines.

- Maintenance code keeps all of the equipment in service, both autonomously and under craftsperson control, and audits data structures to make sure they're healthy.

Administration code is the rest of the craftsperson interface to the Meridian 1. Its "overlays" manipulate the hardware and software configuration database.

To give an idea of the relative scale of these entities, of the 2 million-odd lines of SL-1 code, nearly half are for call processing. Comparing module counts, the subsystems divide up SL-1 like this:



M1 pedants will leap to point out that some PLS subsystems really seem to fit into more than one wedge. Okay, you caught me. The idea of this pie chart is really to give a broad sense of the system composition. If you want more details, you'll just have to keep reading…

What's that Federal Systems slice of the pie?

One of our customers is the United States government, and one of their requirements is that access to the code that controls their software be on a need-to-know basis. Therefore, the whole library is peppered with clauses like:

```
=> IFDEF FEDERAL
=> COPY xxx
=> ENDDEF
```

If you have security clearance, and you set the appropriate context, then the Federal Systems code magically appears in the source. Otherwise, all you can get at is a series of dummy header files. ***Do not disturb*** the IFDEF clauses, or put anything new inside of them!

Sadly, now that you know this, we're going to have to shoot you.

6.1 The SL-1 language

SL-1 is an ALGOL 60 derivative, which is little help to the average new designer today, although people who have worked with Pascal or Modula-2 (or to some extent C or PL/I) will recognize much of what they see. To read SL-1 code, you have to start by learning the language, and the reference manual is as good a place as any to go for that foundation. This section will *not* lament the SL-1 syntax, which is already well explored by the references given in Appendix A.

But to achieve SL-1 fluency, to really understand the code, you need to know much more than the grammar. As a minimum, you'll need to build a vocabulary of important SL-1 symbols; for example, an SL-1 programmer should instantly recognize "`CRPTR`" as the index of the call register that holds the state information for the call receiving the current message, and might remember that on an MC680x0 machine its value is always in register D2. You'll also need to build a mental picture of the software structure. This chapter will attempt to help with both of these tasks.

Turing told us that you could write a given program in more or less any language, but he didn't promise that each would make the job equally easy. By making some things harder than others, the linguistic tools of the SL-1 language shaped how we built the SL-1 system. Above all, there are some the things that *aren't* in the language. It is argued that these omissions were the result of language designers who knew they needed to optimize for execution speed, and did the best they could with limited resources. But this is just an early, special case of the general rule that we should only do in-house those things in which we're prepared to invest enough to do excellently—otherwise we should out-source, especially when there are good enough standard products available.

- There are no run-time integer multiplication or division operators (although there is a `MULT` intrinsic), and no floating point support of any kind. What you can do is add, subtract, and shift left or right. So you'll see code that uses `((A<<2)+A)` because the designer wanted to a speedy way to compute `(A*5)`.

- Scope control for symbols is a bit crude. You can have a system-wide global variable (module `POOL`), a module-wide global variable (local `COMPOOL` declarations), or a stack-based local variable that disappears when you leave the procedure. This situation is improved a bit by SL-1's ability to nest procedures, so that at least you can define a variable as local to a procedure and all the ones nested in it. Even still, good data hiding isn't easy, and there are tons of global variables in our system.

- Lots of common routines have important side-effects via the global variables that they change. This is very speedy, but hard to manage and understand. Expect side-effects any time you see the result of a function call being assigned to `XDUMMY`, or whenever you see a line like this:
  ```
  IF MYFUNC() THEN NULL;
  ```

- Pointers have very limited type-checking. We do a compile-time check that a pointer refers to the right "Target Logical Page" of data (eg: `.U_BASIC`), but on the switch all data is really in a flat address space and we no longer have real paging. On the MC680x0 machines, we reserve register A5 to always point to `SL1MemoryBase`, the start of unprotected data.

- Except for the stack, there are no private variable instances on a per-process basis. The overlay supervisor, which was extended to keep track of the activities of a number of different users, had to go to some trouble to fake this.

- There is no automatic range checking of table indices, although we do truncate to the right field size on assignments. This makes it especially important to do the checks yourself.

- There are no compile-time warnings about the use of uninitialized locals, or about unused locals and parameters. Watch out!

- A standard compiler optimization (called McCarthy evaluation or "early exit") leverages the observation that you don't always need to evaluate every clause in a compound Boolean expression. If A is false, then (A AND B) is always going to be false, regardless of the value of B. Conversely, if A is true, then (A OR B) is always going to be true. SL-1 doesn't do early exit from Boolean expression evaluation. Thus, the following line of real code (from QPRSEG1) is totally unsafe:

  ```
  IF PCFPTR = NIL | LOOPTYPE:PCFPTR ^= .CONF_LOOP THEN RETURN;
  ```

  A safer way to have coded this would be:

  ```
  IF PCFPTR = NIL THEN RETURN;
  IF LOOPTYPE:PCFPTR ^= .CONF_LOOP THEN RETURN;
  ```

  At this point, we're not likely to add this optimization, because (as discussed above) some code relies upon side effects of functions. For example, the following harmless-looking code may have the insidious problem that it requires that MYFUNC always be executed because MYFUNC changes some global data as a side effect of being called.

  ```
  IF (N=0) & MYFUNC(X)
  THEN...
  ```

  If the compiler detected that N^=0 and tried to optimize out the call to MYFUNC, the software would break.

- We don't have generalized dynamic binding or procedure variables, but we do have global procedures that are referenced by an absolute number. This can be used to call different code at different times, most notably with overlays (global procedure #6).

- We don't have a general mechanism for inheritance (making a structure just like some other structure, only with some new bits added on). For fixed variables, the slightly-awkward SET ORIGIN statement is sometimes used to fake this. For relocatable structures, the more common practice is to manage conceptual inheritance manually, by hand-coding the same fields into the same places of each of several structures. For example, all the structures in the DN tree contain XFLAG and those pointer flags, all leaf DN blocks have the DNTYPE field, and all the set datablocks contain a set type field. Similarly, it is critical that the first 19 words of the UTRKBLOCK exactly match those of U_DPNSS_CHAN_BLK, but this is "automated" by a comment warning designers not to mess it up. At least the compiler will choke if you have two structures with the same name but different offsets, so some kinds of accidents will be prevented.

- We only recently got the ability to pass references to structures as parameters, although you always had the option of passing pointers (precisely as fast, but more dangerous...).

- Enumerated types are faked using integers, and the SL-1 idiom equivalent to the C code:

  ```
  enum cookieType {ChocolateChip, PeanutButter, Oatmeal}
  ```

  will normally be:

  ```
  .CHOCOLATE_CHIP = 1;
  .PEANUT_BUTTER = .CHOCOLATE_CHIP + 1;
  .OATMEAL = .PEANUT_BUTTER + 1;

  INTEGER MY_COOKIE (0,2);  % need two bits for a cookie
  ```

  There's no compelling reason not to just hard-code the above using the integers 1, 2, and 3, although doing it as a series does emphasize the fact that you don't really care about the values per se, but only that they be distinguishable from each other. Unfortunately, this abstraction is somewhat defeated by the original CASE syntax, which demanded that you place the code for the symbol with ordinal value *n* in the *n*th clause in the statement. The updated syntax allows the clauses in arbitrary order, but much of the code still out there is written with the original syntax. In general (with the notable exception of definitions to support external protocols) you're using dot constants in a context like this for readability, and not because the underlying values mean anything or are likely to change.

- In days of yore, the design philosophy that smaller procedures were easier to read was intentionally embedded in the compiler as a refusal to accept large procedures. Ironically, this rule was intended to ensure that code was easy to read. As time passed, new designers decided they needed longer procedure bodies, and figured out a way to code around the restriction. That's why you'll see a bunch of procedures called *xxx*_CONT, where *xxx* is the name of the procedure that was a bit too long. The bottom of *xxx* then calls *xxx*_CONT. Logically, just think of these procedures as a single slightly-hard-to-read one.

- The original support for character strings was arguably worse than FORTRAN's, which is something of an accomplishment. Until Release 19, the closest you could get to a string data type was a pair of ASCII characters. Because it was a lot of work for designers to print out helpful messages, a lot of the MMI is a bit cryptic.[24]

6.1.1 "Stupid code tricks"[25]

The following are examples of SL-1 code which work, but are definitely not the best way to do things. Designers who excel in writing such hard-to-read code should apply to the International Obfuscated C Code Contest, which exists specifically to provide a safe arena for horrible code. Others should think of the following examples as anti-patterns.

- ✗ The Camp-On code carefully defines two state constants to have the same value, `.TIME_REM_RECALL=3` and `.CAMPON_RECALL=3`, so that it can reuse a procedure that two different features both need. It then uses both constants, as well as the hard-coded value 3, interchangeably in various places. A better solution would have used the new `CASE` syntax with a "`(.TIME_REM_RECALL, .CAMPON_RECALL): my_proc;`" clause.
- ✗ Various development tools have limitations on the size of the files they can handle, so SL-1 files are usually split into multiple sections, sometimes somewhat arbitrarily. The convention is to have an `MSOURCE` file called, for example, `OVLXXX`. This file consists entirely of "=> `COPY`" commands to bring in the real source, which in turn is stored in `SEGMENT` files called `OVLSE01, OVLSE02, ...`

  Now it turns out each of these files can also include `COPY` commands. It also turns out that there's no check to prevent more than one file from copying the same segment. And in fact, `OVLSE11` and `OVLSE17` both include a "`=> COPY OVLSE19`" record. So if you go look at module `OVL511` in x-view, you'll see two definitions of `PROCEDURE OVLDECIMAL`, and all of the other things in segment 19!
- ✗ The following would code compile, would work, and would not be helpful:

```
IF I:=J+K:=J+3 THEN...
```

[24] People familiar with DMS-10 point out that SL-1 modules support "data" segments, which DMS-10 designers used extensively for handling text messages long before strings were available. For instance, they used this technique to map "LD 11" to "LD DIGITALSETS". I don't know why M1 didn't.

[25] Apologies to David Letterman, and to the designers may have written this code with the best intentions, but without the aids of code inspections, a powerful development toolset, or long-term code ownership.

The equivalent code should be:

```
K := J+3;
I := J+K;
IF I^=0 THEN...
```

✘ The SL-1 language allows a given symbol to be a global, a local, or (perversely) *both.* Within a nested procedure, you can even add an additional local definition of an existing local symbol. Just because it will compile does not make it a good idea. It will usually cause more confusion than it's worth, and it can cause problems that are tough to track down. Try to ensure that your new locals are not already somebody else's existing globals, and vice versa. This goal is made more difficult by the people who made such generic symbols as `I`, `X`, `Y`, `INDEX`, `COUNT`, `DIGIT`, `RESULT`, `STATE` and `PTR` globals.

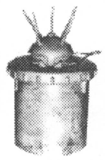
✘ Hard-coded values for global procedure numbers are used liberally, rather than dot constants. "`RETURN VECTABLE[254]^=VECTABLE[.NOGLOBAL_VT_NUM]`" in `OVL511`, which manages to mix the two methods in a single line of source code.

✘ `POOL` contains the definition "`.THREE=3`", whose only usage is in lines like

```
CASE (LOOKUP(.YES_NO_KEYWORDS)-1) OF .THREE...
```

Constants *are* better than scattered magic numbers, but don't apply the rule blindly. `.THREE` doesn't tell you what the constant is supposed to represent (like `.MAX_DIGITS`), and it can never equal anything but "3" without making the code unreadable. Also, it turns out there was only one clause in this particular `CASE` statement, so the programmer should probably just have used a simple `IF...THEN` statement.

A related but better justified example is the family of constants like "`.BUG4001=4001`". While it appears to be at best a waste of time to have defined them, they do facilitate tracking down the source of error messages. The best place to put these definitions is usually `POOL`, but since it is a major hassle to make changes to `POOL`, a number of people have placed them in local `COMPOOLS`, or in `SEGMENTS` that get copied into local `COMPOOLS`. This is tolerable, but if they subsequently get added to `POOL`, the obsolete references should be deleted.

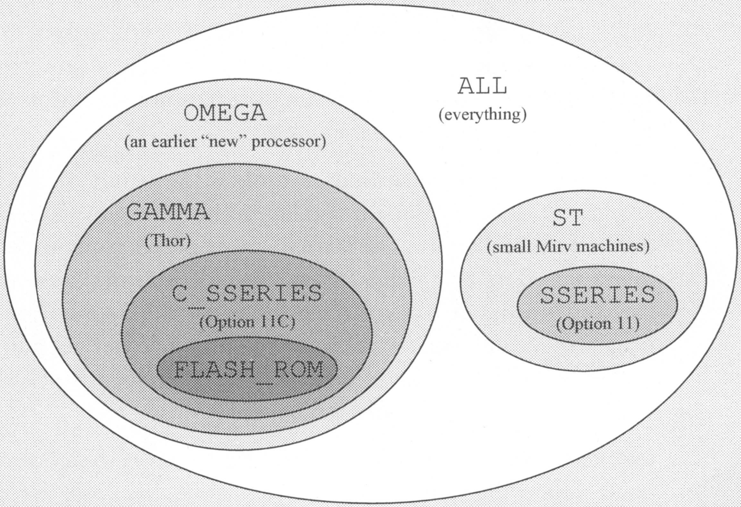


LANDMINE ALERT: There should never be a reason to define any dot constant twice within the same scope. For instance, the code that includes the definition for `.BUG0004` appears twice *within the same segment* of `POOL`. This is an accident waiting to happen...

What are all these `INFOR` & `OUTFOR` tags?

SL-1 has run on a wide variety of processors over the years. By default, most code ends up being shipped to all target machines, with the compiler bearing the brunt of the porting workload. But in cases where machine-specific code is required, the `INFOR` and `OUTFOR` compiler directives let designers specify which target loads will get their code. `INFOR` means put it in; `OUTFOR` means leave it out. The available choices are:

- `ALL` = everything
- `OMEGA` = the 24-bit AMD bit slice CPU. `INFOR OMEGA` clauses also get included in loads for both Thor & Option 11C, so for just Omega you need to say "`INFOR OMEGA & ^GAMMA`".
- `GAMMA` = Thor, the Motorola MC680x0 family of CPUs, including Option 11C
- `C_SSERIES` = Option 11C, including flash machines
- `FLASH_ROM` = Option 11C machines which execute from flash memory (for speed), rather than shadowed DRAM. Somewhat confusingly, this directive does *not* include the later generations of Thor (CP2/CP3) which also use Flash ROM.
- `ST` = "Small Turbo", a Mirv machine, the 16-bit AMD bit slice CPU small system (predating Omega). `INFOR ST` is also included in Option 11 loads, so for just ST machines you need to say "`INFOR ST & ^SSERIES`".
- `SSERIES` = Small Series (Option 11)
- `MSERIES`, `SL1M`, `N_OR_XN` = *ancient* sections. These should no longer appear in live code.
- `PROTOTYPE`, `DEBUG` = test code, could be anything, should not go to the field



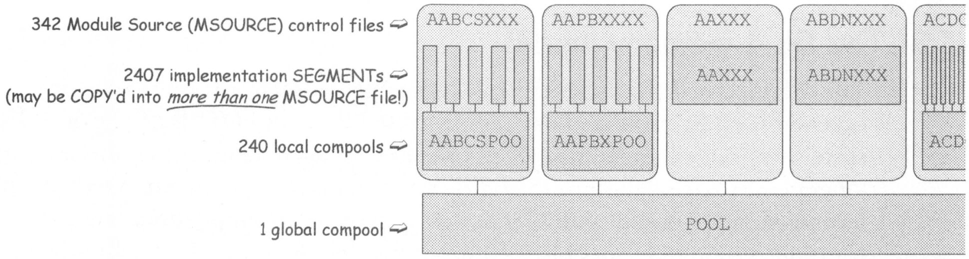
6.1.2 SL-1 code structure

It's a dynamic design, which is a polite way of saying that you're going to have to go read the code if you really need to know exactly how anything works at any given time on any given machine, but this chapter can still help you understand what to expect.

The whole design philosophy is bottom-up: "I can cluster together variables into a structure, and group procedures into a module" versus "I can implement this abstract data type by having a number of fields in this structure, and break down this module's functionality into a number of separate procedures". There is an overriding embedded systems mentality visible in the core SL-1 modules. Coding is close to the machine, with little or no abstraction. It is painstakingly optimized at various points for either store usage, or execution speed, or both.

There are exactly three tiers in the static SL-1 code hierarchy. The lowest, POOL, is the mother of all APIs: 90,000 lines of code which declare *every* symbol that is shared by two or more SL-1 modules[26]. Regrettably, every variable, constant, and procedure declared in POOL is then accessible to every module in the system. The middle tier of SL-1 is the collection of local COMPOOL segments, which declare the shared symbols for a particular module. On top of both of these are the main implementation segments. In some cases, (eg: module AAXXX) the middle and top tiers are combined, and the local COMPOOL appears in the same PLS segment as the code.

Static SL-1 Code Organization


342 Module Source (MSOURCE) control files
2407 implementation SEGMENTs
(may be COPY'd into more than one MSOURCE file!)
240 local compools
1 global compool
AABCSXXX
AAPBXXXX
AAXXX
ABDNXXX
ACDC
AAXXX
ABDNXXX
AABCSPOO
AAPBXPOO
ACD
POOL

[26] I'm lying. It's not quite true that every shared constant is in POOL. The other, scarier, technique is to have each module that needs the constant declare its own local copy. Now, as long as they're all the same, things will work out fine. The only safe way to achieve this is put the definition into a PLS segment, and then COPY that segment every time you need it. Alternatively, you have to hope you don't get blown away by typos. Finally, you'll see the occasional muddle whereby all three techniques are employed in different places to define a single constant. This is a bad thing.

Why `COPY` procedure segments more than once?

Each `COPY` adds a copy of the machine code for your procedure into the M1 program store. `COPY` provides a convenient way to get around the old PLS limitation of 1000 lines per module. But memory is still not free, although it's now getting cheaper. Why would more than one module ever `COPY` the same segment?

Historical reason #1:

"Overlays" used to be overlaid. That is, there was a single overlay space that, at any given time, could have exactly one of the various overlay modules in it. Any code that was common to more than one overlay was best handled through a single shared segment that each overlay could `COPY`.

Historical reason #2:

There used to be a strict limit to the number of global procedure numbers available. If you had several copies of the executable code, one in each module that needed to call it, you didn't need to use up a global procedure number.

Dodgy pseudo-OO reason (👎):

Since `COPY` is essentially a macro expansion, the symbols the copied segment refers to will be specific to the context into which it is copied. Thus, you could design polymorphic behavior into your segment by defining different local copies of the procedures the copied procedure calls. You could also sort of get inheritance by referencing different kinds of local variables along with some shared structures. But the odds of people successfully maintaining code like this, even assuming you could get it right to begin with, would not be high.

6.1.3 The SL-1 memory model

The original SL-1 machine had 4 × 16K pages of memory: one each for executable code, protected data, unprotected data, and I/O buffer space. Ever since then, our pointers have been based on a paged model, even though it no longer bears any relation to the hardware. The technique might eventually be leveraged into a decent virtual memory system, or to support memory protection for multi-tasking, or possibly to support private variables. But right now it isn't.

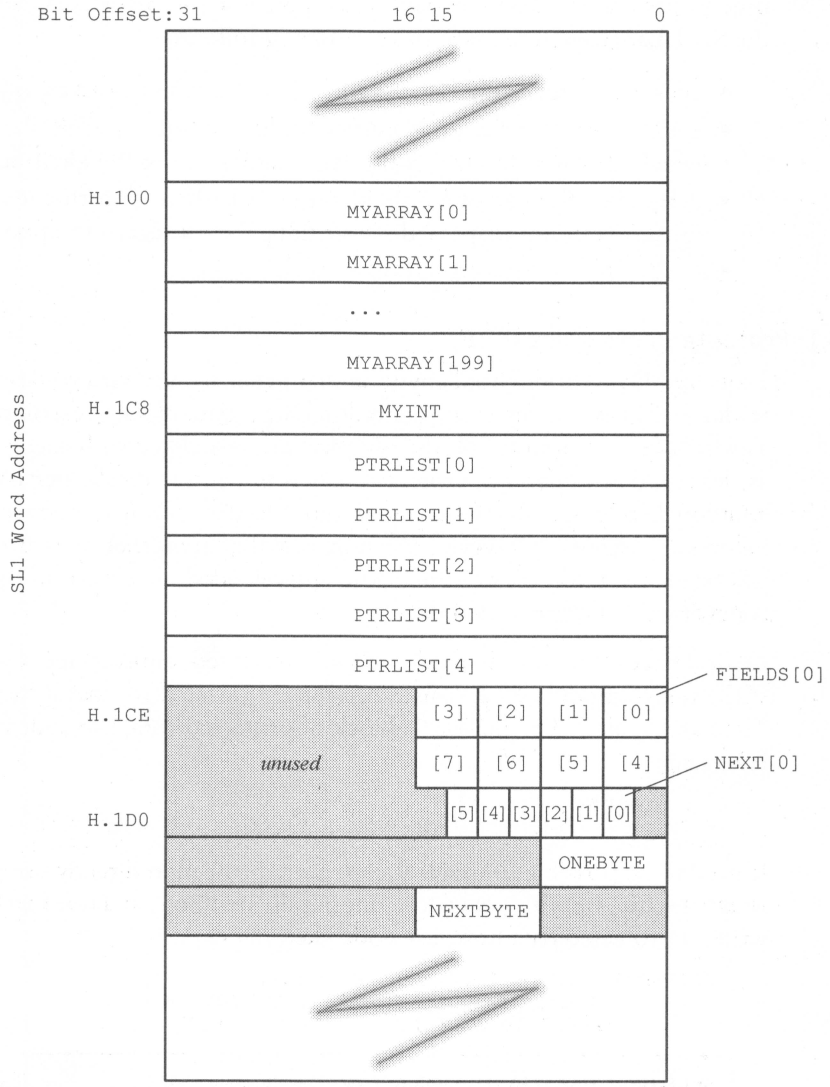
We have what amounts to a virtual machine implemented half at compile time and half at run time, which maps from SL-1 addresses to native target-machine addresses. Regardless of the conventions of the target CPU, SL-1 addresses start bit counting with the low-order bit. There is also a slight residual propensity to think all things are 16 bits long, although between the Omega (24-bit) and Thor (32-bit) ports, most of these have been cleaned up. The key details are:

- For arrays of little things (bit fields), the compiler only fills the low-order words. This is mostly because some of the old code, that explicitly steps through structures and does its own pointer manipulation, would fail if we fix this.
- On a DATADUMP, only low-order words are saved. On SYSLOAD, only the low-order words are restored, with the top half being set to zero, rather than sign-extended. Thus the DN tree has to be re-derived from the terminal blocks (rather than just being restored).
- Pointers are composed of two parts. The first has evolved from a 2-bit physical bank-switched page number into a 9-bit logical page number which is used for compile-time pointer checking. The second is the "SL-1 address", an offset that has evolved from 16 to 32 bits. To get the real native pointer, you have to shift the SL-1 address two bits to the right, and then add a base offset. However, as long as the CPU speed is much faster than the memory fetch time, it probably doesn't really cost us anything to do this run-time indexing.
- Type `INTEGER` now uses the full 32 bits of a longword. It is no longer safe to assume `#FFFF` is the same as `-1`.
- Packing is big-endian, although we're still puzzling about how best to handle the Intel port. Within the CPU it doesn't matter much, but as was pointed out earlier the M1 machine is really a network of processors, and the messaging can get tricky if the CPU is byte-transsexual.
- There's an added barrier to understanding, which is that the syntax differs slightly between "structures" and global variables. With global variables, the assumption is that each variable will begin on the word boundary following the previous variable, and you use the `SET ORIGIN` command to override this. Since structures are basically type definitions, it doesn't make any sense to try to reset the "origin" in the middle of them. Also, structures tend to be things that get repeated many times in memory (think of Call Registers), so people worry more about how to pack them, and almost always use the syntax where each field's starting offset is specified individually. The array packing rules are still the same.

An example may help to visualize the SL-1 memory model, and will also make obvious the potential memory savings that future compiler work might yield. Assume the following lines of SL-1 code:

```
COMPOOL
BEGIN
   SET ORIGIN H.100;
   INTEGER MYARRAY[200];
   INTEGER MYINT;
   UPOINTER PTRLIST [5];
   INTEGER FIELDS (0,4) [8];
   INTEGER NEXT (2,2) [6];
   INTEGER ONEBYTE (0,8);
   INTEGER NEXTBYTE (8,8);
END;
```

Given the above source code, the compiler will actually build the variables in memory like this:



Note especially that `NEXTBYTE` does not land in the same word as `ONEBYTE` without resetting the origin.

To help people write portable code, SL-1 provides a family of "pseudo-procedures" which are guaranteed to return the right portions of a data structure irrespective of the CPU on which they're running. Pseudo-procedures generate in-line code rather than procedure calls, and have the magical characteristic of being able to "run" at compile time (if enough information is available) or run time (if passed relocatable variables as parameters). The exhaustive list is given in the SL-1 Language reference, but a sampling follows:

- `ADDRESS` - returns the "SL-1 address" (not a native pointer) of its argument
- `LOGICAL_PAGE` - takes an identifier, and returns the Logical Page Number, of which the high-order bit indicates whether or not the identifier is protected
- `BITWIDTH` - the size of a bitfield in bits (1 to 16). Unfortunately, this intrinsic can not be passed bigger things like integers and pointers.
- `SIZE` – the size of a structure, in words

6.1.3.1 Protected data store (PDS)

Protected Data should hold anything that needs to survive a restart. This is traditionally the customer configuration data—data blocks describing things such as which features and telephone numbers are associated with each set. Note that, as discussed in Chapter 2, protected data is not automatically persistent; additional code must be written to dump it to disk, if it is to survive power failures or reboots. However, it is normally the intent that most protected data will be archived to disk in some form, and reloaded on a reboot. By contrast, this is never true of unprotected data.

Logical pages 0 to 511 are hard-coded as protected, unprotected, or non-existent at the top of module `POOL`, usually referenced by tag ("`.U_ROUTE_DATA`", "`.P_BASIC`", etc.) To "malloc" a block of protected data, use code like the following:

```
P_MQA_PTR :=
GET_PDATA_BLK(SIZE(MQA_DATA_BLK),LOGICAL_PAGE(MQA_DATA_BLK));
```

It used to be possible to toggle the memory protection directly using `PROTECT(.ON)`, but this procedure doesn't do anything on Thor machines. To write to protected data now, use code like:

```
WRITEPDS(LOGICAL_PAGE(ACD_AGENT_ID),
         ADDRESS(ACD_AGENT_ID:P_POSITION_PTR),
         AGENT_ID_CODE);
```

> **WARNING:** Overlay 43 (Datadump) locks out all other overlays, so most of the risks of writing to PDS during the dump are avoided. If you're writing to PDS from call processing, you will be okay as long as all the associated writes are done during a single timeslice. Otherwise, you could cause an inconsistency in the dumped image. *Reading* from protected data store does not require any special effort.

6.1.3.2 Unprotected data store (UDS)

This is where all transient data should go, most notably call registers and state information for anything that doesn't survive restarts. Information like who's talking to whom, which lamps are lit, and which phones are ringing, is stored in UDS. Note that call registers don't really survive warm restarts; they are reconstructed as well as we can manage out of the network connection memory. A lot of the more subtle data gets lost in the process.

Medical History

We keep Call Forward on No Answer (CFNA) data in protected store, but because we let people change it frequently, we kept basic Call Forwarding (CFW) data in unprotected store. Some guy (we'll call him Dr. Marcus Welby, M.D.) had his phone forwarded to his beeper. Now one day, the switch did a restart. It recovered normally, but of course it lost the CFW data in the process. The next thing that happened was a medical emergency, in which it would have been awfully helpful to have paged that guy whose phone was no longer forwarded.

We've since moved CFW data to Protected Data Store.

Moral: Your choice of data store type matters more than you might think. End-users shouldn't know about restarts and power losses, so the things end-users do care about should persist across such interruptions.

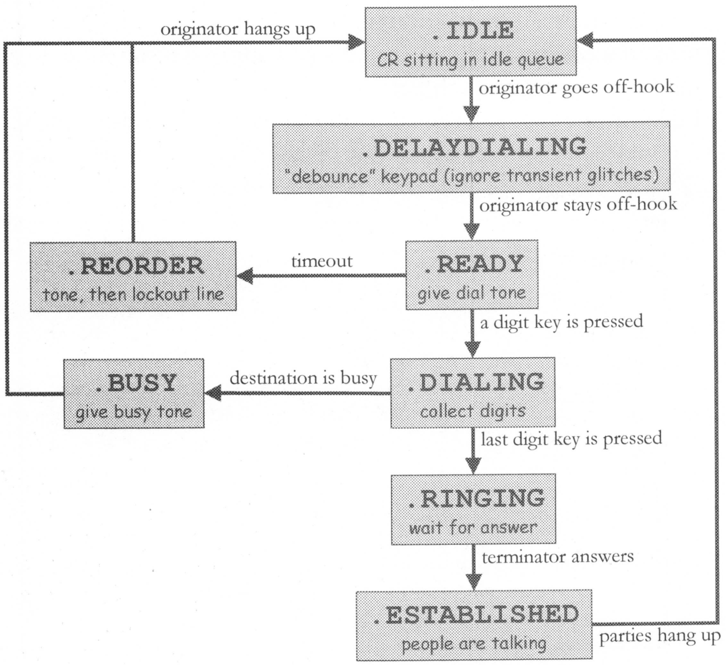
6.1.4 Major SL-1 data structures

The reason the M1 mostly works is that all of the code that manipulates call control and terminal data structures lives in SL-1 and is scheduled by `WORKSHED`. All designers should have at least a basic understanding of the data structures discussed in the following sections.

6.1.4.1 Call Registers and CRPTR

Call Registers (CRs) hold the information about each active call. They contain such things as who originated the call and to whom the call was made, what time the call started, which features are active on the call, what digits were dialed, and the current call state (in `MAINPM` and `AUXPM`).

`MAINPM` typically goes through the following transitions:



SON_CRs

Even though CRs are getting kind of big (about 180 words[27]), there isn't room to store a lot of the specialized feature data in a regular CR. If we made the CR big enough, then all calls would have the extra overhead. The compromise chosen was to allow a linked list of up to 52 extra data blocks ("son" Call Registers) to be linked on as needed. Many features define their own refinements to the `SON_CR` structure, and link them to the main CR using `CREATE_SON`. They cost a little more real-time to get to, but they make sense as long as most calls don't need them. An alternative is bit reuse: overlay two or more semantic meanings on the same CR fields. This works well, but forces the features to be mutually exclusive.

CRs as a generalized memory abstraction

CRs are also used by some features as placeholders in timing queues, or even as print buffers. Because we had already gone to the trouble of providing a fast way

27 That's right. We've got nearly 1K bytes of complex, interconnected, *overlaid*, state data, and we still need to link extra `SONCR`s. Getting call processing code right is tough: it takes real skill, and thorough testing.

of allocating and freeing CRs, and auditing them to make sure we never lost any, reusing them as a generic buffer is kind of a reasonable thing to do, provided that you don't hold on to them too long, and that you make suitable changes to the engineering recommendations so the switch doesn't run out of CRs. It does mean that we end up using medium-sized buffers to hold small things, but as long as we're not fragmenting the memory this is not a huge problem.

You can look at the number of CRs in each queue on a live system by using the pdt command "`sllqShow`". On a very slow switch, the output might look like this:

```
SL1 queues of size > 0
   Cadence (queue  2 at 0x4ab186c) size : 2
   128LowP (queue  3 at 0x4ab1880) size : 14
      2Sec (queue  4 at 0x4ab1894) size : 19
      Ring (queue  5 at 0x4ab18a8) size : 5
      Dial (queue  6 at 0x4ab18bc) size : 6
      Idle (queue 12 at 0x4ab1934) size : 4460
       RAN (queue 14 at 0x4ab195c) size : 2
```

6.1.4.2 Terminal Numbers (TNs)

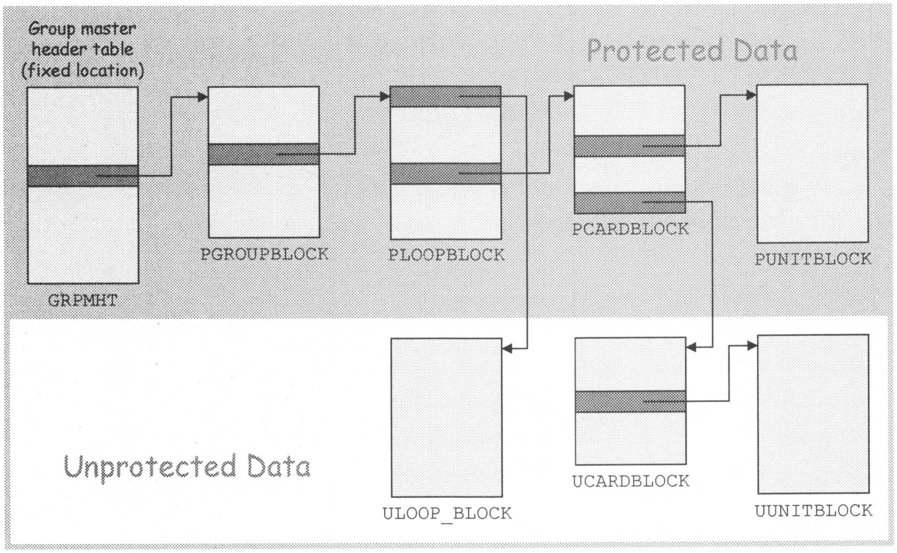
The usual software abstraction for the telephones, trunks, and service circuits involved in a call is the TN. The TN actually specifies the physical location—the network group, loop, shelf, card, and unit—where this terminal's wire terminates on the PBX. Because it is so closely tied to the hardware, the internal format used for TNs depends on the vintage of the machine. Originally, we could handle only 4 telephones per line card. Over the years, this has evolved to up to 32 sets today, but the size of the TN is still 16 bits, thanks to slightly hideous packing sorcery. There's also a special version for Option 11C, since it has no use for shelf or card information. Fortunately, almost all call processing code can treat the TN as a handle, a unique identifier for the phone in question, without attempting to parse the internal structure.

Group, Loop, Card, and Unit Blocks

These blocks form a dynamically-allocated tree structure that holds all of the protected configuration data and the unprotected state data for each TN on the switch. It also holds the data for each component in the hierarchy (group, loop, shelf, card, and unit). `TNTRANS` is the intrinsic normally called to traverse the tree, and it initializes a group of pointers, one to each of these blocks. By convention, the local variable which points to this pointer group is called an `ITEMPTR`.

Note that the protected items point to the unprotected ones, and never vice-versa, so that you get the right behavior over restarts.

The TN Tree



`TDBLOCK` and `CONFBLOCK` are the analogous structures to link tone detectors and conference bridges, respectively, to call registers.

TN templates

In the beginning, memory was very expensive. Therefore, designers were always trying to think up ways to limit the amount of memory used by the M1 system.

One of the ways that they came up with, was to separate the protected TN blocks (for both PBX and BCS sets) into two areas:

1. the "fixed area" that contains data that's always needed, or data that is not big enough to warrant the allocation of a dynamic entry which will use up at least one word of memory, and
2. the "dynamic area" which allocates memory only when required by a dynamic entry, dependent on a TN's configuration.

Information in the fixed area can easily be access using a fixed data structure since all of the data is fixed. However, the information in the dynamic area can vary from TN to TN, depending on what dynamic entries are configured. Therefore, a fixed data structure cannot be used for this dynamic area.

Instead of allocating extra memory to store the type and size of each and every dynamic entry in a TN's dynamic area, the idea of templates was used instead.

The assumption was that most TNs would have the exact same dynamic configuration as many other TNs in terms of sizes and types (not content).

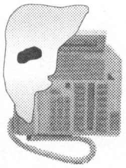
Therefore, templates were created. These templates define the type, size and location of the data that is stored in the (dynamic) "template area". Each TN block contains an index (for faster access, a pointer was also later allocated) to the appropriate template that defines this "template area". In this way, the template can be used for the template area, similar to how the fixed area's fixed data structure is used to access the appropriate data. Therefore, any TN with the same template area configuration can share the same template.

For example, if a TN were created, and copied 1000 times, only one template would be needed. The fixed area content (eg: DN, CFW DN, HUNT DN, etc.) can be different, but the same template can be shared as long as the type and size of each template entry is the same.

Model telephones

The above templates are instantiated automagically when new TNs are created on large systems. The Option 11C makes the same idea accessible to the craftsperson by providing a wide variety of pre-programmed, model telephone layouts from which to choose. Using telephone layouts or templates, technicians can perform a few simple steps at installation to activate multiple telephones.

6.1.4.3 Phantom TNs (PHTNs)



M1 Terminal Numbers took a step towards greater abstraction in Release 20 with the invention of Phantom TNs. For some applications, it is convenient to have a terminal profile that is *not* linked to a physical phone. A Phantom TN (PHTN) is ordinary terminal configuration data for a phone that does not physically exist. This allows customers to define TNs and associated DNs without buying the hardware associated with them (i.e. phone sets, line cards, etc.).

For example, we allow ACD agents to "roam" to a different desk each day. When the agent logs on, we automatically invoke Remote Call Forwarding to transfer all her calls to today's real DN. PHTNs also allow multiple published DNs to terminate (again via Call Forwarding) on a single real phone, which can be convenient in certain service industries. Yet another use for these Phantom TNs was for defining templates for DTEV terminal provisioning. The craftsperson defines a series of "models" with typical user profiles, and then uses these for auto installation.

To ensure we always send calls somewhere, the feature that created PHTNs also needed to create a default Call Forwarding DN. In the absence of other instructions, a call to a PHTN will go to this default DN.

We have developed different flavors of PHTN over the years. The original Phantom TN was based on the 500 set, and can only be used as a temporary home from which to forward calls. No hardware is required for these, but the customer does have to use Overlay 17 or 97 to tell the M1 that some loop number is the "phantom loop", and then all TNs on this loop will be PHTNs. With Incremental Service Management (ISM), we charge people for the combined number of real TNs plus PHTNs configured (although it's sort of selling software by the pound…)

The second type of PHTNs was built on top of standard BCS set call processing code for an early mobility product needed to be able to launch calls from a PHTN. For instance, a user may give an assortment of destinations where he or she might be located, and we could attempt calls to each of them to try to track down the real person. These BCS-type PHTNs were reused for Controlled DN processing by Symposium. Because these PHTNs are used to originate calls, it was expedient to require them to be associated with a dedicated physical network loop. The alternative would have been to visit all of the lower-level code being invoked and teach it that it not to try to connect speech paths and send messages to the non-existent phones.

This is exactly what a later mobility project did. Now calling them "Virtual TNs", the project extended the 500-type PHTNs to microcellular sets. A portable set often changes its physical location (hence, "portable") and thus its access point to the M1 network. Virtual TNs are used to store the portable set's configuration. Like the original PHTNs, Virtual TNs require no physical hardware. But Virtual TNs are used throughout call processing code, so that Mobility users have access to the full suite of M1 features. To achieve this the PHTNs were changed from disabled to enabled for call processing. On microcellular calls, real physical TNs (ports on an MXC card) are only used to reserve network paths for connecting speech path. Virtual TNs allow mobility to handle the many-to-many mapping of roaming sets, and the extra layer of concentration from idle terminals. The model looks like it may extend well to IP telephony.

Yet another type of PHTN, built on top of DTI2 trunks, is used by Digital Private Network Signaling System (DPNSS) code. DTI2 PHTNs are used for ISDN Semi-Permanent Connections, for Australia.

These various PHTN types have been layered on top of each other over the years, and the resulting code is not always too clean. For example, each loop can either host real TNs or one of the different PHTN types, but instead of having a single table recording which of these it is, there are three tables, `PHTNLOOP[]`, `PHTNBCSLOOPS[]`, and `PHTNDTI2LOOPS[]`, which are used to try to track this single state. Tracing through the use of these tables reveals a lot about how the Phantom TN code works. In particular, be aware that code that appears to apply to all PHTN types, such as function `IT_IS_A_PHANTOM_LOOP()`, may only apply to a subset. It would be helpful to merge these some day.

There's also another, unrelated fake TN type called Logical TN (LTN). It was created for Integrated Message Systems (IMS) voice mail in Release 14, and is used to map each IMS attendant terminal to a voice messaging port. And these LTNs are not related to the Meridian Evolution Logical TNs, nor to ISDN Logical Terminal Identifiers (LTIDs).

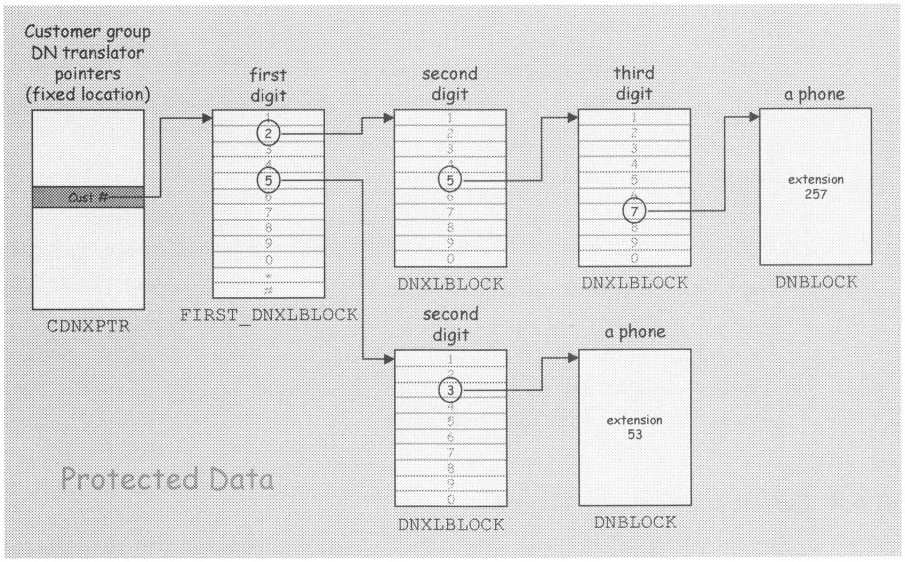
6.1.4.4 Directory Number blocks

Directory Number blocks (`DNXLBLOCK`) form a dynamically-allocated tree structure that helps call processing software route calls based on the digits dialed. The main features of this tree are that it supports variable length dialplans, and that it may be traversed very quickly by call processing code. `DNTRANS` is the intrinsic normally called to do this traversal.

Because customers all have their own dialplans, `CDNXPTR[`*`cust#`*`]` is used to find the root of each DN tree. Each branch of a DN tree either terminates in a `DNBLOCK` leaf, usually corresponding to a local telephone, or grafts onto a `ROUTE_MASTER_HT`, which leads to a list of trunks routing out of the PBX. As with the TN tree, each node in the tree is a relocatable structure. The simplest PBX sets need no further data, so no `DNBLOCK` is even allocated.

An additional tree for each customer group is used to handle Flexible Feature Codes, and this one is rooted at `FFC_CDNXPTR:P_CUST_DATA_BLK`.

The DN Tree



Chapter 4 of the M1 Core Course notes contains a thorough discussion of both DN and TN translation. The notes are available on the web at: http://47.49.0.148/Department/Training/course_documentation/

Controlled DNs (CDNs)

A CDN is used for off-board call control (like SCCS), and is basically an ACD Directory Number without agents. There are two modes: default and controlled. In controlled mode, the CDN acts as a parking lot for calls waiting for treatments from the controlling applications. In default mode, which is normally only used if the AML link fails, it works like ACD.

CDN calls are controlled by messages coming over the AML (either ELAN or LAPB). When the call is first terminated on the CDN, an Incoming call (ICC) message is sent over the AML to the call-control application. This application will then return a message that might tell the M1 to connect this call to music, route it to a DN, or even merge it with another call.

It is common to publish several different numbers for subscribers to dial that correspond to different services. Once the call reaches the M1, there are a variety of mechanisms that can remember how the call arrived (known as Dialed Number Information Service (DNIS)) before placing them all in the same CDN queue. The DNIS data is then included in the ICC message so that the call control application may decide how best to handle the call.

Merge call is useful for applications that want to send their call to a destination, but want to make sure it terminates before connecting to call. Meanwhile callers can "enjoy" recorded announcements, music, ringback, IVR, or even silence while in CDN queue. Multiple calls can be made to different destinations on behalf of the caller in the CDN queue (as in PCS), and then the CDN call will be merged with the first successful one.

CDN implementation reused much of the ACD code, which made it easy to provide the queuing and default ACD behavior.

Virtual DNs (VDNs)

Virtual Network Services require a pool of VDNs to track calls. Up to 4000 VDNs may be configure using Overlay 79. One is required for each active VNS call.

6.1.4.5 VECTABLE

`VECTABLE` is an array, indexed by global procedure number, of the start of program store for each of these procedures. It is used to locate procedure bodies when executing the code, and for SNAP and PDT. On some machine types, `VECTABLE` is also used for packaging, by overwriting entries for packaged-out procedures with a nil procedure, combined with setting the appropriate bit in `SERV_PACK_RESTR`.

6.1.5 Some common hooks

Calls to the following procedures are scattered liberally throughout the SL-1 library. It may be helpful to understand what they do, and when to use them.

6.1.5.1 SNAPVAR()

There are nearly 1200 calls to `SNAPVAR` sprinkled throughout the call processing code. These calls trigger data collection for a tool called SNAP. SNAP traces the execution of the SL-1 code without stopping it, and also displays useful details about key variables like `MAINPM:CRPTR`.

SNAP output is helpful, but it tends to be a bit verbose. There is a post-processing tool on Unix called "roadmap" which thins out some of the redundant information, formats the output, and best of all substitutes the local procedure names where appropriate to give a much more readable result. The following is an example of the kind of output roadmap produces.

```
WORKSHED : input_task
. LIN500 : lin500
.   LIN500 : disconnect_msg
.   . ONHOOK : onhook
.   .   DISCONNECT : disconnect
.   .   . DISCONNECT : camp_on
.   .   .   CAMP_SEARCH : camp_search
.   .   .   . MUSIC_MODULE : music_module
.   .   .   . CAMP_SEARCH : rem_percamp_tone
.   .   .   .   REMOVE_SON : find_son
.   .   .   .   REMOVECRPTR : unlink
.   .   .   .   .  REMOVE : removecrptr
.   .   .   .   .  DECR_ATTN_QU : decr_attn_qu
.   .   .   . DIGPROC : digproc
.   .   .   .   DIGPROC : dialing
etc.
```

SNAP is extremely valuable in the early stages of debugging a problem. The only downside is that you have to manually code the calls to SNAPVAR, or it won't do anything. A lot of effort has been spent embedding these calls into most existing features, but beware that there may be places that were missed.

A complete user guide can be found under F02907, *SNAP Tool Slides*, in Doctool library MLVDOCS. From Unix, "man roadmap" also gives quite usable instructions.

6.1.5.2 BUG()

When call processing software detects information which is not in the correct format or gets into an invalid state, a BUG message is output. BUG messages are intended to assist designers debug their code. Ideally, BUG messages should never occur in the final product.

BUG numbers are managed globally to ensure each is unique. To get the next available number, send an email to "SL1 MSG (BNR)".

6.1.5.3 ERR()

General hardware or database problems are reported with ERR messages. These problems can be corrected in the field (eg: a hardware failure or database configuration error). If possible, instead of an ERR report, use the error type related to your hardware or software component, eg: Primary Rate Interface (PRI) and Remote Peripheral Equipment (RPE).

Unless errors are suppressed, this global procedure prints "ERR" followed by your error number. Since Release 19, BUG and ERR messages normally get filtered along with all the other error messages by the centralized fault management reporter.

6.1.5.4 WARNING()

This is *not* a global. It's one of dozens of local procedures of this name, with many unrelated implementations, which puts out a warning message on the attendant console's display, or to memory, or to a TTY screen, or whatever the local designer wanted when it was written. Don't use this without checking what it means in your context.

6.1.5.5 MARKOVERFLOW() and MARKBUSY()

These routines update the call state in the CR, provide the right tone to the caller, notify the appropriate services, and peg the operational measurements when a call is unable to complete.

6.2 The PBX platform

6.2.1 Sysload

As stated at the beginning of this chapter, SYSLOAD is the first SL-1 procedure to be invoked after a reboot. It's main job is to load the most recently dumped customer configuration database into protected memory. SYSLOAD is a big procedure (group of procedures really), but in simple terms, SYSLOAD sets up the protected data, and then calls INITIALIZE to set up the unprotected data. Only after this has been done will call processing begin.

Datadump is controlled by LD 43 & LD 143 overlays, and can either be part of the automated daily routine or done manually. The main reason customers do regular Datadumps is to recover gracefully from power failures or severe memory corruptions.

When a customer upgrades from one release to the next, all of the protected data must be converted to reflect any changes to data structures between the releases. This job is done by procedure CONV, which is invoked by SYSLOAD. To a designer, this essentially means that you have to write a new procedure, which will use WRITE_PDATA to spit out the new version of your structures, and then link your new procedure into the right procedure called by `X81CONV`. It's ***really important*** to get this stuff right, because customers rely on being able to reboot in cases of emergency. They're already in a bad mood if they have to reboot. They get even less pleased if the reboot doesn't work.

WARNING: Datadump only dumps the low-order 16 bits of your 32-bit data.

For more information on what you need to do to make sure protected data for your application is converted correctly, see B01826, *SYSLOAD and CONVERSION*, in Doctool library BVWDOCS.

6.2.2 Initialize

The `INITIALIZE` procedure is invoked for one of three reasons:

(a) VxWorks is doing a cold restart, for instance because the system has just rebooted.

(b) VxWorks is doing a warm restart, for instance because a hardware failure in the common equipment has occurred, or the manual initialize button on the common equipment has been pressed.

(c) VxWorks has initiated an SL-1 task restart or the `tSL1` has trapped (although from Release 22 onwards, this just causes a warm restart).

The operations performed differ slightly according to the cause of the restart, but in general the following sequence is executed:

- Rebuild the unprotected software data blocks: queue structures, I/O data blocks, Customer Data Blocks, Route Data Blocks, and TN blocks. Allocate the Call Registers and place them in the idle queue.
- Test most devices for response and for permanent interrupts. Disable and mark as faulty any that do not respond. If an interrupt cannot be cleared by disabling the offending device, then mask out that interrupt. If the initialize was caused by a failure of an I/O device then mark that device as faulty.
- In redundant systems, if any faults seem to be present but the inactive side seems to be healthy, then choose which side looks most promising to run on.
- Perform or initiate downloading to X-cards such as XNET, XPEC, and XCT, as well as certain set types (XNPD, MISP, and MSDL).

- Invoke REBUILDCALLS to rebuild the call registers for calls that were established before the SYSLOAD or INITIALIZE occurred. Connections to either conference loops or TDS (or MF-sender) loops are not rebuilt and are cleared in hardware.
- Enable interrupts so that message processing can recommence.
- Print messages on all Maintenance TTYs giving the details of the restart, and start Call Detail Recording.

6.2.3 Switchover

On redundant machines (that is, machines with twin CPUs) the active CPU does all of the work, and the inactive CPU just waits around in case the first fails. The hardware ensures that the data store on the inactive side shadows that on the active side, so that in the event of failure, the inactive CPU can jump in instantly and continue.

Now that our operating system has been disentangled from the SL-1 code, switchover is really a platform concern. The only real requirement is that SL-1 provide suitable initialization code to be invoked on an ungraceful switchover.

6.2.4 Workshed

As discussed in the Patterns chapter, WORKSHED is the overall work scheduler for the SL-1 code during normal PBX operation. It's really a Transaction Engine manager, making sure that the call processing, administration, and maintenance code all gets to run fairly, and that they do not end up trampling each other.

The first priority of WORKSHED is call processing messages. There is a philosophy that under heavy traffic "committed" calls, that is calls that are already in progress, should take precedence over new calls, so we defer processing origination-type messages until other types of calls-in-progress messages have been handled. If all of these have been handled, WORKSHED will do administration work, and then background work like tone cadences, audits and timers are serviced in successively lower-priority "tiers". Recent work has been done to give AML call message handling a similar priority to more traditional call processing messages.

Note that interrupts are asynchronous events that can happen at any time during the above scheduling. However, all that should usually happen during interrupt handling is that a message be appended to the appropriate queue, which will then be processed in order when WORKSHED gets to it.

6.2.4.1 Timeslices

The key concept to understanding WORKSHED is the timeslice.

Consider a (slightly simplified) phone call:

- a caller goes off hook and receives dialtone
- he presses a digit, dialtone stops, and the PBX remembers the digit
- he presses another digit, and the PBX remembers the digit...
- after enough digits, the PBX translates the digits and rings a destination set
- the destination answers, and the PBX connects speech path
- a caller disconnects, and the PBX takes down the call

Each of these steps is a single, atomic transaction. Any number of events may make demands on the PBX between or even during (in the case of interrupts) each step. Other calls may come and go; users may log in or out; the time of day will change. But it is very convenient to be able to ignore all of that and just concentrate on this call as if it were the only thing happening.

Each of the above steps corresponds to a WORKSHED timeslice. During each, a message arrives, an action is performed, and state data is updated so that the call processing code will know what to do with the next message. WORKSHED ensures that no other SL-1 code runs during your timeslice. In return, you have to promise to finish what you're doing and return fairly quickly (under about 100 milliseconds). Since no other SL-1 code will run until you finish, various real-time critical jobs are being delayed, and eventually we'll start dropping trunks. The watchdog timer will detect gross abuse (after two seconds) and restart the switch, but you will have started causing trouble long before this. Also, if you're really running long transactions very often, you'll be seriously impacting the total call capacity of the switch before long.

Now consider a database change. This time the transaction processing happens at many levels. At the simplest level, a user presses a key, the PBX remembers the keystroke, echoes it to the terminal, and waits for another keystroke. At a higher level, these characters assemble into command lines, and the overlay processor interprets these one at a time, modifies the database accordingly, and awaits the next command. Overlay input processing is done in between call processing transactions, so it will occasionally change the value of a structure that matters (like the DN tree) during a call. But this will never happen during another *transaction.*

So when does my code get to run?

There are essentially three ways code can be invoked.

1. **Interrupt code:** Interrupts signal the CPU directly that they need immediate attention. They get to run almost immediately, but are restricted to very short amounts of work. In general, all an interrupt handler will do is to enqueue a message to be processed by task-level code.

2. **VxWorks task code:** This is the normal execution mode for all non-SL-1 software, including the VxWorks kernel itself. It is strictly priority based, meaning that the highest priority task that has work to do is always the one running. It doesn't really have a transactional "timeslice" paradigm, although it can share things equally within a priority level.

3. **SL-1 task code:** One of the VxWorks tasks is `tSL1`. All SL-1 code runs under this task, and to manage this SL-1 runs its own scheduler (`WORKSHED`) which is designed to let transactions on the call processing data complete atomically. For instance, once we start to process an off-hook message, no other SL-1 requests will be handled until we've found and filled in a call register, updated the terminal state, and queued for dial-tone. This helps keep the state data sound in the face of the barrage of messages that hits the switch during heavy traffic. `tSL1` usually runs at a very low priority, and so is interrupted by most other tasks. That's why it's important that other code normally avoids changing (or even examining) call processing data structures. Within `tSL1`, there are a number of effective priorities, but the key is that they don't interrupt each other within a given transaction. If a higher priority message comes in, the previous transaction still completes and then `NEXT_TASK` finds the highest priority transaction waiting to run.

6.2.4.2 Timers

As discussed many times, SL-1 code is transaction based. In general, you can't place a `DELAY(10 seconds)` in the middle of your transaction, because you would hold up the rest of the world. What you do instead is to set a timer, and then ask `WORKSHED` to let you know when the timer expires.

Basically you have a choice between two levels of timer granularity: timing "ticks" can be either 128 milliseconds or 2 seconds. There are many variations, but the general idea is to set either (or both) of `ORIGTO` or `TERTO`, the originating and terminating timeout counters in your call register, to the right number of ticks, and then link the CR into one of the timing queues. `WORKSHED` invokes `TIMING_TASK` every 128 milliseconds, which decrements the counters appropriately. Whenever one of the counters reaches 1, `WORKSHED` calls your code, based on how you've set `MAINPM`. If your feature needs to run extra timers, grab a free call register or a timing block, link it to your call, set a counter for the number of ticks you want to wait, and then link that block into one of the timing queues.

There are a few important things to note here. The first is that the timing is only approximate. Depending on where you are within a tick when you first ask for the timer, you could lose almost the entire first tick worth of delay. At the other end of the wait, once the timer expires, you will only be invoked as soon as WORKSHED gets to you. If there are other messages waiting, or if other timers timed out at the same time as yours, there may be some delay. The other important thing to stress is that your task is not suspended. You will normally continue to handle other messages during the time you're waiting for your timer. If one of those messages means that you no longer have any need for the timer, you may cancel it by UNLINKing the CR from the timing queue. Finally, whenever you LINK your CR to a timing queue, you can wait forever by passing the value .TIMER_OFF, or continue the countdown on any existing timer by passing in .UNCHANGE.

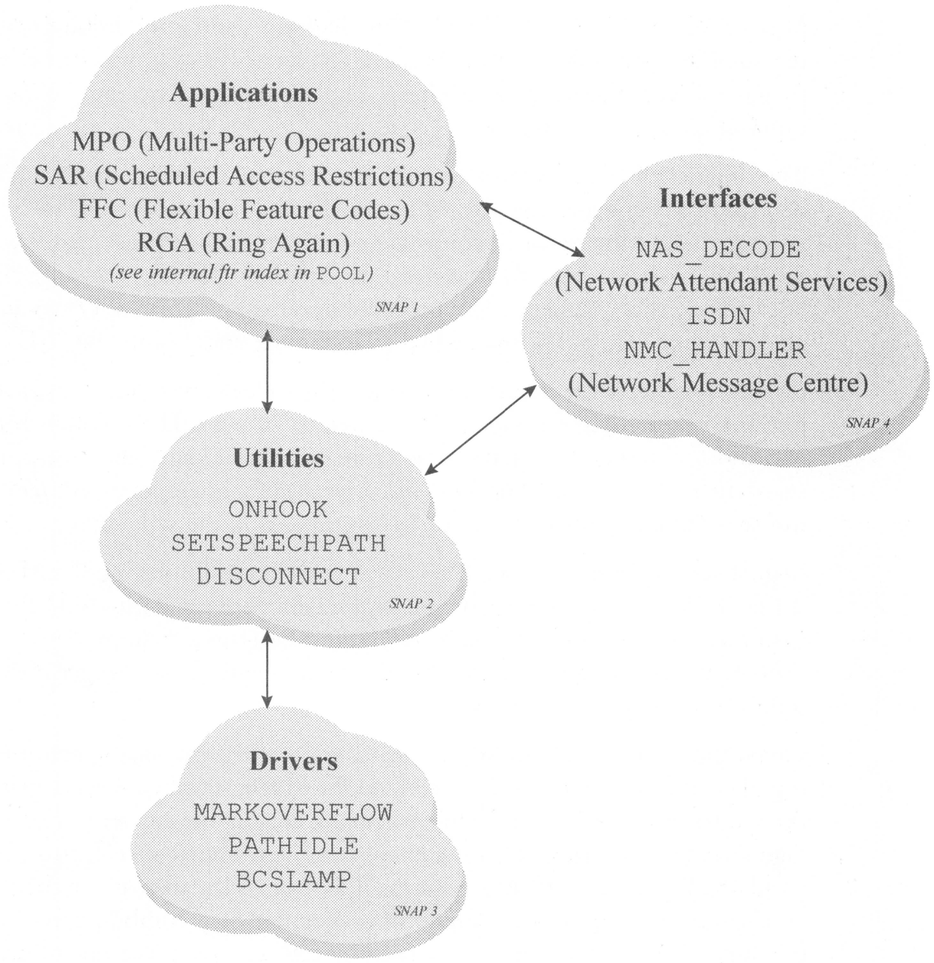
6.2.5 I/O

The Meridian 1 has gone through many incarnations. Because each transition was done with limited resources, and because we needed to have a smooth transition strategy for our installed base of switches, large parts of the system were usually left unchanged from one generation to the next. In particular, you will notice that we still use a Tape Emulation system (TEMU) for reading and writing to mass storage, even though the medium has changed.

For sending characters to TTY terminals, use the global procedure LOGPUTCHAR, or higher-level output routines like DECIMAL.

6.3 Call processing

The call processing code comprises roughly a million lines of SL-1 software. It was written by hundreds of people over a period of more than twenty years at half a dozen different sites around the world, and so it would be unfair to claim that there's a very complete uniformity of design thinking behind it. Nonetheless, learning how any given feature works is quite a tractable piece of work. To begin with, there is a certain rough structure to the code as a whole:



The above structure is not rigidly adhered to, but you can find it in the code. These days, most of the utilities and drivers that a new feature needs are likely to exist already, so the exercise becomes one of piecing together the required functionality from a collection of useful parts (sort of like Dr. Frankenstein did ☺).

[A future issue might attempt a decent annotated catalog of call events and utilities.]

6.3.1 Messaging

The original Peripheral Equipment did analog-to-digital conversion on the speech path, and simple concentration (by virtue of the fact that at any given time most phones were idle). The PE had no "intelligence", and any stimulus on a phone (eg: digit key pressed, digit key released, receiver off-hook) was just relayed up to the CP. This made peripherals cheap, but led to the CP needing to handle a large number of extremely simple messages with minimal delay for each one.

The original analog line cards had a Scan and Signal Distributor (SSD) chip that detected state changes for up to 4 terminals, built an appropriate 16-bit message, and signaled to the network controller's terminal scan process that it was ready to send the message. The SL-1 sets and the original attendant consoles also had SSD chips inside of them. The protocol became known as SSD signaling. An analog line-to-line call normally had 12 incoming and 4 outgoing SSD messages.

As new features were developed, particularly caller name and number display, the number of outgoing messages skyrocketed to 70 per call! One reason for this enormous count is that each character in a display update has to be put into a separate SSD message. Another is that we seem to send an inordinate number of messages to ensure that the handsfree speaker is in the right state.

Digital sets are connected via Time Compression Multiplexing (TCM) loops. When TCM was introduced, its messages were converted into SSD format to help the evergreen-ness of the Network and Peripheral Equipment. So even though we had an opportunity to start working with fewer, longer messages, it didn't really work out that way.

Given the history, it shouldn't be surprising that the messaging complexity comes together in module `DSET` (which created the "Delta II" digital sets), inside procedure `TCM_OUTPUT_MSG`. This procedure is used for any sort of control of a digital terminal. It may request a change to the terminal speech paths (eg: turn on handsfree), update an LCD key status indicator or the display, or change the set configuration. The particular request is specified by a combination of two parameters. To get a sense of the possible range, look at the raft of message types called `.CMD_`*`xxx`* or `.DCON_`*`xxx`* in module `POOL`. In all cases, once the TCM message has been formed, it is sent to the line card (originally an ISDLC, now probably an XDLC) via the PBX output buffer using `SEND_PBX`. The buffer contents ultimately get converted into SSD messages by the intrinsic `WRITE_ENET_NWK` or `WRITE_XPE_NWK`.

The SSD message then goes out to the XNET controller, which relays is to the XPEC on the PE shelf, which converts the SSD messages *back* into TCM signaling for the benefit of the line card and terminal!

6.4 Operations

Operations, Administration, Maintenance, & Provisioning (OAM&P) is the vintage telecom industry term which includes almost all of the non-call handling things our customers need to run their switches. These functions are not really as cleanly separated as my section-headings make them appear, so please bear with me as I try to cover most of the essentials.

Under Operations, I will focus on two groups of operating data an M1 produces: traffic and billing.

6.4.1 Traffic

Traffic statistics, also known as Operational Measurements (OMs), are used to help make engineering decisions: Do I need to buy more trunks? Is my CPU fast enough? Which networks are getting the highest usage?

Traffic measures things like the number of times all trunks are busy, high-water marks for resource usage, and device failures. The usual term is to "peg" an OM, meaning to increment that particular counter when a condition is detected. Every half-hour, the accumulated counts are transferred to "hold" registers and tested against critical thresholds, and a subset of the traffic measures are printed.

Traffic data is kept in unprotected store, in data blocks like ULOOPBLOCK and UTRKBLOCK. It may be printed in reports by Overlay 2, and may also trigger alarms at preset thresholds. See especially modules TFC and TFP.

6.4.2 Billing

Call Detail Recording (CDR) is our per-call billing system, such as it is. CDRs can be used to track who called whom, which features were invoked, and how long each call lasted. If enabled, CDR software will send a stream of 220-byte records, typically one for each trunk call, over a serial data port to a downstream billing processor. Billing systems sometimes also view CDRs statistically (like OMs), to track patterns in overall call rates.

Because we take an interrupt to push out each byte of each CDR, there is a substantial overhead for the CPU. CDRs are often turned off because in many PBX environments there is no attempt made to bill users, and generating CDRs just puts an unnecessary load on the system.

CDRs are buffered in Call Registers in `.QU_CDR` while they are waiting to be sent out the serial port (another example of using CRs as a generic managed memory system). However, call processing will steal back unprocessed CRs from `.QU_CDR` if no other CRs are free. Under heavy pressure, a PBX would rather provide dial tone than guarantee to bill for it.

Many different types of CDR may be produced, depending on whether the call completed normally, what sort of services were invoked, what sort of facilities were involved in the call, etc. For more details, try the *X11 Software Features Guide*, which dedicates 192 pages to CDRs.

6.5 Administration

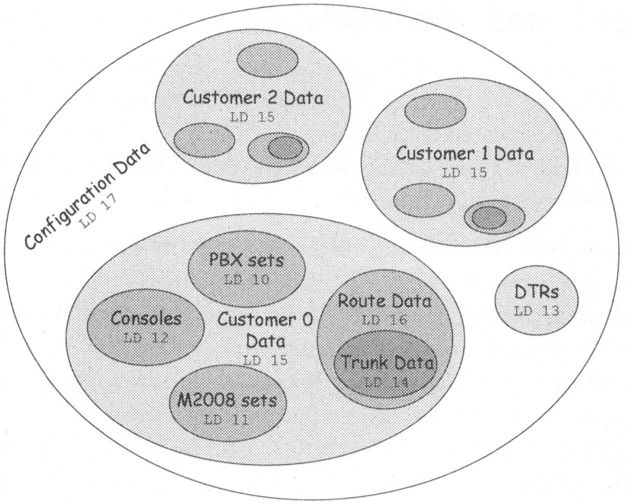
6.5.1 Overlays

The first thing to understand about "overlays" is that we don't overlay them anymore. In the earliest SL-1 machines, we only had a total of 16K to load *all* of the executable code, so we used a fairly standard technique to cheat a bit. We knew that there was only one craftsperson terminal (now false), and that they could only ever want to be working with one area of configuration data at a time (false even then), so we split the MMI code up into a series of files. Each file could manipulate one of the main areas of data, and we would load one of these files at a time into the single shared 1K "overlay" space, sandwiched between the 8K ROM and the real start of Program Store. This is the origin of the name overlay, and the reason you enter an overlay by typing "LD" (for "Load").

Overlays do one of three basic jobs:

- "Service Change" tools, prompt-driven tools which change configuration data for the system, customers, terminals, features, routes, etc.,
- "Print Overlays" which display the above data, or
- "Maintenance Overlays" which manage the other components of the SL-1 system.

The structure of the permanent switch data is nested, and must be entered in the right sequence. For instance, because each dialplan is specific to a customer, you can't enter route data before you've defined the customer to which it will belong. The diagram below tries to show how the main pieces fit together.



For multi-customer M1s, we actually treat the switch as if it were split, and you end up needing to use real (albeit *very* short) trunks to route a call between customers. There's no abstraction in the MMI of a dialplan as distinct from route selection or physical trunks, although there is one built internally (the DN tree).

The job of overlays usually involves updating some element in the protected data. Since this is generally dangerous, and since we allow people to type "****" to escape out of any activity they decide they aren't happy with, the usual coding technique is to work with a temporary buffer called a `WORKAREA`, and only copy the results to protected memory when we're sure the craftsperson wants the change.

6.5.1.1 Linked overlays

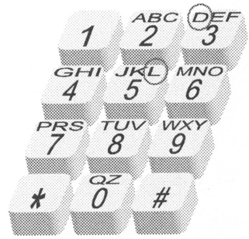
Because we originally only had room for one overlay at a time in main memory, the user interface used to force a craftsperson to leave one overlay to perform actions in another. In particular, if you defined a PBX set in overlay 10, you had to quit it and enter overlay 20 to print out your data. To reduce this headache, we have now linked a few of these together (at last check, overlays 10, 11, and 20, but *not* 12, 13, or 14) so that you can get at all their commands from any of them.

You can recognize that you are in a "linked" overlay because the prompt is "`REQ:`" instead of "`REQ`". It might be argued that there is still much room for improvement, although the real solution is to use MAT.

The complete set of overlays is described in the *X11 Input/Output Guide* (four volumes), or is available on-line at http://47.82.33.147/~mtvjbg01/SL1Overlay/SL1OverlayIndex.html.

6.5.2 Set-based administration

On very small M1s, people may not want the expense of a TTY terminal. Therefore, we allow an ordinary PBX set to be used to configure the database. Sometimes installers will bring a TTY with them to set the switch up initially, and then take it away with them when they're finished. Other real masochists might in theory do the whole thing from a phone.



Apart from the obvious difference that the signal is coming in from a phone, there is the other equally obvious difference that a phone doesn't have big 104-key PC keyboard. In order to reuse almost all of the software, we have carefully chosen each of the command names in the overlays to map uniquely to the corresponding telephone digits. A procedure (LOOKUP) in the overlay manager maps "LD" to "53", because "L" is on the 5 key, and "D" is on the 3 key of a telephone. Because of the way this was done, it is actually possible on a TTY to type "KE", "JE", "LF", or any other letter pair that maps to "53" instead of "LD" (if this sort of thing amuses you…). The other non-TTY approach is console-based administration. However, these days neither of these methods is used very often, and many of the newer overlays do not really support them. The real trend is to move all OA&M to MAT.

6.5.3 The overlay supervisor

The overlay supervisor allows users to log in, and then it used to have to load in each bit of overlay code whenever someone typed "LD *xx*". Since all code is now always in memory, this overlaying of HMI code is no longer necessary.

However, overlays were originally implemented using a set of global variables to keep track of what *the* craftsperson was doing. Because you can now have more than one craftsperson terminal on an M1, the overlay manager must manage to keep these right for whichever terminal has sent in a given request. `SET_SLICE_VARS()` does this for multi-user administration.

Also, a small number of overlays can be nested. This is known as "overlay linking", and is also managed by the overlay supervisor.

6.5.4 Security

A craftsperson must go through a login procedure to get access to OA&M functions. **Limited Access Passwords** (LAPW) give users access to specific overlays or debugging tools, depending on the user level. Some may have "print only" access. Multi-customer switches support an extra level of security to keep customers from interfering with each other. Up to 100 userids and passwords may be configured using Overlay 17. An audit trail may also be requested, so that you can later tell what each user did (or tried to do!).

Many auxiliary processing subsystems, including Meridian Mail, ICCM, MAT, and MICB, also have there own independent security arrangements.

SL-1 call processing can restrict access to toll trunks using **Code Restriction**, administered by Overlays 19 and 49. The authorization codes used then get recorded in the associated CDRs, so that bills can be traced back to the user.

6.6 Maintenance

For M1, see G00021, *OA&M Standards for Meridian 1 Developers* in Doctool library GLOBPROC.
For Meridian Evolution, see M02253, the *Fault Management Developer's Guide* in Doctool library SL1DOCS.

6.6.1 System Event and Error Reports (SEER)

SEER, written for the Alarm Centralization project, has added a fault management filter on top of the traditional SL-1 error messages. It provides a standard interface for managing all of these, and automatically escalates severity with repeated failures.

Each error message has:

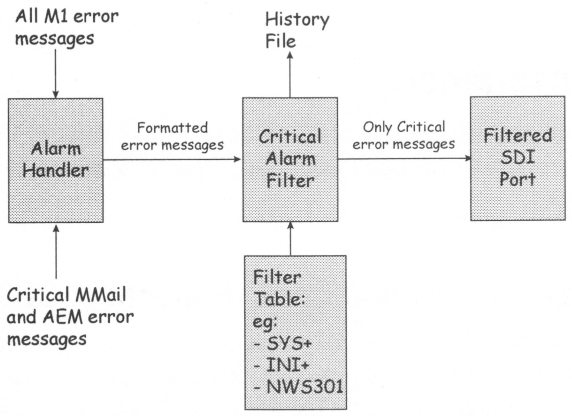
- a mnemonic, for example:
    - ERR (for operational errors, such as buffer overflows, but also used for some bugs)
    - BUG (software errors, such as # ACD agents < 0)
    - SCH (Service Change, usually typos by admin folks)
    - AUD (audit noticed a problem)

  - SYS (mostly packaging issues, and some general errors. The SYS alarm numbers are in hexadecimal because they used to be printed by code that didn't have access to decimal conversion routines.)
  - etc.
- a four digit alarm code, unique within this mnemonic
- a severity, `.SEER_CRITICAL`, `.SEER_MAJOR`, `.SEER_MINOR`, or `.SEER_NONE`

For a discussion of the code, see M01076, *Meridian 1 Fault Management* in Doctool library SL1DOCS. For a complete explanation of all 14,000 messages, see http://47.74.128.167/stj/Msgs01.html.

6.6.2 Alarms

Critical errors (like dead trunks) cause alarms to be raised at the site. We have a general filtering system which goes something like this:



6.6.3 Overload controls

The SL-1 maintenance software attempts to detect, report, and isolate "babbling" cards. If we notice a component spewing out too many messages within a given time window, or if a large percentage of its messages are invalid, we assume it's a babbler. The overload software also kicks in if we run out of input buffers.

The software tries to decide if the problem is with a unit, card, shelf, or loop, and to disable the smallest component possible. It also warns the craftsperson by spitting out an OVD log.

6.6.4 Audits

`MAIN_AUDIT` and `LAMP_AUDIT` were already mentioned in Chapter 3's discussion of the Corrective Audit pattern. They are substantial collections of code which are responsible for ensuring that the presumed state of various SL-1 entities matches the actual state.

7. Management

7.1 What's in the job description?

Classical telecommunications has a sub-discipline called Operations, Administration, and Maintenance (OA&M), or occasionally OAM&P, where P=Provisioning. This is basically all the rest of the applications-level code that doesn't do the call processing. By their nature, these functions are easiest to do with in-depth knowledge of both the platform and the call machine, so the SL-1 software is not unusual in having some of its OA&M code strewn throughout the rest of the system. In fact, "manageability" probably qualifies as one of the important pervasive aspects of the system that Chapter 2 examined. Like the other aspects, it too tends to apply to all components and all levels of granularity. The difference is that there's also a substantial corresponding software structure to support the relatively small local hooks.

The new and improved ISO jargon for OA&M is Fault, Configuration, Accounting, Performance, & Security (FCAPS). FCAPS captures the nature of the job slightly better, but both terms are still common.

7.1.1 Fault management

The first duty of the management code is to deal with faults. Fault Management code must isolate the error down to the smallest field-replaceable unit, alert the maintenance personnel that something is amiss, and then tell Configuration Management to take the misbehaving equipment out of service and determine whether there is any other path to subtending equipment or if it too must be taken out of service. At a manned site with an organized spare equipment inventory, we expect the Mean Time to Repair (MTTR) to be around 45 minutes or better.

A particular challenge is that we must be able to diagnose a problem that has caused the system to restart or even reboot (and hence lose all its state information). To do this, the platform allocates a report record in a special part of memory that is preserved across reboots, and the error handling code tries to write some meaningful notes here before restarting things. Although this doesn't help recover from the fault directly, it may provide useful debugging information.

7.1.2 Configuration management

Another obvious job of the management subsystem is to allow a craftsperson to alter the configuration of the equipment. More recent systems may even automate most of this task ("plug-and-play").

7.1.3 Accounting

At the high level, we need to track which features are installed, and how many telephones are equipped, because this determines what PBX owners pay us.

At a more detailed level, we record authorization code usage against long distance calls in Call Detail Records (CDRs) because this allows PBX owners to distribute their costs appropriately within their user community.

7.1.4 Performance

We need to report things like how many calls we are handling during busy hour, how many times a trunk group is completely busy, which equipment had problems, and which system bottlenecks are approaching their limits. This allows our customers to plan when they will need to upgrade their equipment.

We also need to detect and handle overload conditions.

7.1.5 Security

We must control access to the system as a whole, and to certain functions (like intrusive debugging tools) in particular. This is even more important now that the M1 supports LAN and dial-up connections.

Within call processing software, "authcodes" help customers control costs by providing restricted access to toll trunks.

Security may also interact with Configuration Management to ensure that customers get what they pay for (and not more).

7.2 Legacy OA&M

7.2.1 The data architecture problem

M1 contains a complex database built by people whose expertise was not in complex databases, and whose principal focus was call processing. It has evolved over a number of decades, and is now somewhat labyrinthine and arcane. It has little tricks like storing its 32-bit entities in a 16-bit file system, and using TNTRANS to infer semantic links between disjoint structures. Our new user interface (MAT) hides some of its blemishes, but one is still tempted to pull the whole thing out and replace it with a good, commercial database. This would make archiving, redundancy, integrity semaphores, and version control somebody else's problem (say, Oracle or Sybase). It would simplify the design job of load-sharing the call processing task across processors. It would reduce maintenance effort, and probably make training new designers easier.

And it would be a mammoth task, have no immediate customer benefits, and probably lead to serious code bloat and real-time problems. For now, we're not seriously pursuing this idea, and on-switch management tools must come to terms with the existing code. The only promising alternative that the MAT team is attempting to leverage is to wrap the existing code to present a fairly normalized database façade to the external management box(es).

7.2.2 Overlays

The overlays which handle most of the traditional SL-1 OA&M have already been discussed in the preceding chapter, but I will briefly reemphasize a few key ideas here.

OA&M code is not typically real-time critical, and shouldn't be where a PBX spends much effort during a busy hour. Therefore, we originally stored much of the administration code on a tape and loaded parts of it as needed into a common overlay space. These "overlays" are permanently resident today, but the name and much of the design have carried over from earlier days.

Part of the reason they are still the foundation of even our more advanced OA&M system is that they need to modify feature and equipment configuration data, and the only safe place to do that is within the SL-1 transaction engine (WORKSHED). If translations or provisioning data were allowed to change at random times, it would cause havoc to call processing.

The rest of the reason we still use overlays is that they were written on top of POOL, which allows them to peer inside of the call processing system. Recent code uses inheritance to present a standardized management API to the rest of the world (discussed in the next chapter). Older code sort of didn't need to, because OA&M code could always see everything it needed to directly. This was fast, but the tight coupling made evolution harder.

7.2.3 Hardware Infrastructure (HI)

HI is the maintenance infrastructure of the M1. It's either the highest level of the platform, or the lowest level of the management software, depending on your politics.

On earlier machines, any bus error would cause a warm restart. The M1 system had 10 shelves and 30' cables worth of extended addressing, so there were lots of opportunities to drop bits. It was clear that the software would need to detect bus errors reliably and at least retry the operations, but before Thor, the maintenance code and the operational code of the SL-1 were intertwined in a kind of Gordian knot.

Hardware Infrastructure was envisioned to separate out that code which allows the machine to reason about itself and hardware failures. It focused on the physical layout of the system, with no "logical processor" abstraction. Some configuration and maintenance overlays were merged, effectively adding a standardized management portal to all pieces of equipment, with a uniform enable/disable/test/configure interface.

The HI *task* maintains CP hardware information, decides on the severity of error conditions, and issues switchover or task restart decisions. Additional HI *interrupt* code captures information at the time of bus errors and sends it to the HI task for handling.

HI maintains and relies upon a self-descriptive information model, driven by a number of control files:

- `hi.db` defines all Hardware Infrastructure files and thresholds
- `cnib.db` describes all the CNI cards in the system

- `cp.db` describes the Call Processor cards (2 typically) in the system
- `iop.db` describes IOP cards (2 typically) in the system
- `ipb.db` describes connections between what network group and which CNI port
- `simm.db` describes how many SIMMs are installed on each CP card.

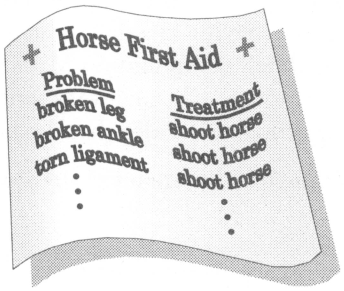
It provides a simple state tree, reflecting the physical cabling of the network, and file system redundancy. The Dependency Management part of HI might have (but doesn't) propagate state changes downward, so it is not really as useful as it could have been.

7.2.3.1 Trouble "shooting"

When a fault happens, we hope the following activities will happen:

1. detect the fault
2. collect information
3. diagnosis the real problem
4. form a strategy to solve the problem
5. choose the best tactics within that strategy
6. plan a procedure
7. execute that procedure (clear the fault)
8. selectively escalate certain recurring/important problems

Machines are good enough at 1, 2, and maybe 3. Beyond that, we'd prefer to hand the job over to humans, except that most of our systems are unattended. Therefore, we at least need to have a default response, to try to restore some measure of service until a craftsperson can get there. There's an old Far Side cartoon with a veterinary student memorizing a list that went something like this:



We do something similar. HI has a table with pre-arranged treatments for problems, with the most common result being to kill call processing. Of course, in our case the system doesn't stay dead—it comes back to life through some kind of restart. But, also of course, shooting the horse doesn't fix the broken leg. Sometimes a restart, or even a reboot, does not clear a problem. If it looks like a problem with redundant hardware, HI may try to switch to the other side. If it is in a single pack, like a tone circuit, HI may try to take that hardware out of service and carry on as well as it can.

7.3 Trends in management

The Information Technology industry keeps moving the yardsticks, continually increasing customers' expectations of functionality (if not reliability). Our system management designers, in particular, must keep abreast of these trends.

7.3.1 Automation

In the earliest switching systems, if something failed to work, the craftsperson might have been alerted by the smell of an electrical coil melting itself. He'd then pull out a circuit tester, and go around hunting for the fault, looking for loose cables, feeling for overheating components, checking for moths in the relays ☺… the system wouldn't give many clues, but then, it wasn't too complicated either, and the problem would get solved.

As systems have evolved, we tried to add a measure of automation to the process. This includes automated configuration, fault detection, and repair (or at least isolation) of faulty equipment. The trend is towards having the craftsperson specify policies and processes for management, and having the machine take care of itself. The switch has a model of its pieces, and the dependencies between them. The different parts of the FCAPS framework then work together to achieve the desired behavior. Fault Management may notice a component having trouble. FM would tell Configuration Management to update the state of that component in the model. CM might take information from Accounting and Performance Reporting to decide what the best workaround would be.

We should continue to take whatever steps we can in this direction.

7.3.2 Networks

In the first SL-1, there was only one processor, and it was on the core shelf. Now, we have multiple local CPUs (CP, IOC, INS, NCs, ACCs, EIMC, MISP, and assorted Application Modules) networked together within the M1 system. Our customers might wish to manage multiple systems within a campus, perhaps including SL-100s or even routers in a coordinated fashion. And a large corporate or carrier customer might wish to manage the wide area network, doing things like dynamic route control or network ACD load balancing to handle traffic changes or outages.

Our management strategy must move customers from element management to network management, and beyond this to business process management based on process and policy decisions. Getting there probably means tiers of management boxes, with the management systems themselves being manageable by other management systems.

7.3.3 Management Information Bases (MIBs)

If you have OO technology, or even careful traditional design, all of the elements can be made similar, by having the same state space and similar abstract operations (enable, disable, test, fault-diagnose, fault-propagate). The "tough part" of management is data attribute administration across all object types. The problem is that it is driven by the implementation details of all of these different object types, which may be somewhat difficult to abstract.

With OO, the Common Object Request Broker Architecture (CORBA) allows objects (and the management thereof) to be distributed among processing nodes. CORBA uses Interface Definition Language (IDL) is the base mechanism for object interaction, and specifies remote procedure calls for object management across platforms. But CORBA is really about full-scale object interaction, and not just the remote control of managed objects.

The generic, not-necessarily-OO, network management version of this is called a Management Information Base (MIB). In the world of Simple Network Management Protocol (SNMP), a MIB contains the definition of all managed objects on a network element, and provides a framework for identifying and managing the objects within the network. The MIB accepts commands from the managing system, and sends out a stream of state information about the managed objects. The MIB comprises a hierarchical database, and a table-driven API.

Before it was cancelled, ME had two example MIBs. A MIB on the Fore INS stored all configuration data, which operations were permitted, and any alarms, and was managed via SNMP get/set and traps. On the CP node, the System Infrastructure included our own home-built MIB.

7.3.4 Graphical User Interfaces (GUIs)

When the cryptic command line interface was introduced, it looked like an enhancement compared to LEDs and front panel toggle switches, or debugging by sense of smell. These days, people expect (possibly 3D, full-color, animated, hyper-linked) graphical representations of the important information, and help screens and/or intuitive interfaces that allow them to leave the manuals in their original shrink-wrap.

This all takes processing power, which is a scarce resource on the Call Processor platform. Therefore, we introduced the System Management Platform (SMP), whose role is to convey the internal details to a craftsperson in a way which lives up to these expectations. Where the management code used to reside entirely within the CP, it is now largely off-board. However, there is still heavy interaction between the on-switch management framework and Meridian Administrative Tools (MAT), the software running on the SMP.

7.4 Meridian Administrative Tools (MAT)

MAT consists of a Windows 95 or NT-based GUI and a set of applications used to manage Meridian systems. Over time, the existing character-based overlays will be used less and less, as the GUIs incorporate more functionality. This should reduce training costs, speed up operations, and eliminate some kinds of errors in user input.

MAT offers different user interfaces depending on the experience level of the user: skilled maintenance personnel, skilled configuration personnel, or clerical users doing bulk data entry. The Enhanced Command Line Interface will also still be supported via Overlay Passthrough. Context-sensitive help screens are provided where possible.

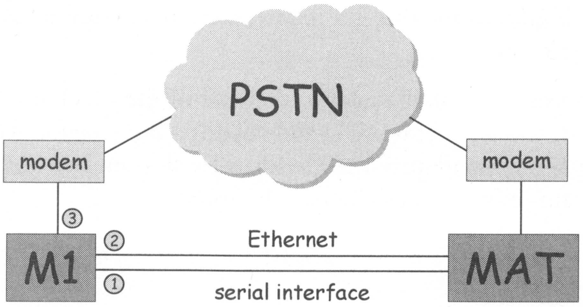
MAT 6 is basically MAT 5 with all of the appropriate changes to support the Meridian Evolution hardware. Added to this, there is an express provisioning mode, automatic card detection and in-service, Peripheral Software Download (PSDL) control, a more uniform state model, an inventory tool, and a substantial reduction in the number of overlays to be used. The overall goal of the program is simplification of system management. Presumably, it will need substantial rework once the new evolution strategy is finalized.

MAT differs significantly from previous management products in that it is task oriented. That is, MAT presents the end user with coherent, consistent steps oriented towards completing whatever task a user needs to execute. The goal is to significantly enhance end-user productivity, and actively reduce complexity and confusion.

There are at least two markets that may demand the long-term support of the command line interface, in addition to any GUI we might build. For Option 11C and especially 11C Compact, the total system cost is low enough that adding a PC for management may make us uncompetitive. At the opposite end of the spectrum, power users may have their own off-board boxes to control many different communications systems (especially things like Cisco routers). These systems will typically be built around scripts that drive the command line interface, and do not extend easily to support a GUI. Therefore, although MAT is very important for some customers, it can't be our whole management story.

7.4.1 MAT ↔ Meridian 1 communications

MAT has a choice of physical interfaces to the M1. For local connections, the physical connection was traditionally an RS-232 serial data link ①, and is now usually Ethernet ②. The Ethernet connecting MAT with the M1 is known as the embedded Local Area Network (ELAN). The ELAN will also extend to the Symposium Call Center Server (SCCS) and the InterNode Switch (INS). It is critical that a router be used to separate the ELAN from the rest of the customer LAN, to protect each from the traffic loads of the other. To access remote sites, we would typically run PPP over a modem link ③.



Across any of these links, we use a suite of higher-level protocols.

The old command line interface was telnet-based "TTY" emulation, and this is used even on the latest version of MAT to get at some of the older overlays, like Station Administration and ESN configuration. Even some of the newer GUIs are really just front-ends for the old SL-1 overlays.

Alarms are signaled to the Event Monitor application using Simple Network Management Protocol (SNMP) over User Datagram Protocol (UDP). We also use SNMP for the session management code that controls logon. We have defined Alarm and Session MIBs to control the SNMP transactions.

For ME, MAT also ran a separate SNMP session directly to the INS, using Fore's ATM Management Interface (AMI). In the future, this might be done through the M1. Database archives and software delivery use Network File System (NFS) over UDP. Finally, request/response style transactions for switch administration in the Equipment Management application for ME would have used the new Management Interface (MI) software, built on TCP/IP. MI combines an Attribute Dictionary and an `enum` conversion service to build a mostly version-independent message transport service over RogueWave sockets (`Net.h++`). For coding expediency, the MI service is now also used by Overlay 117 code even if the old command line interface is being used.

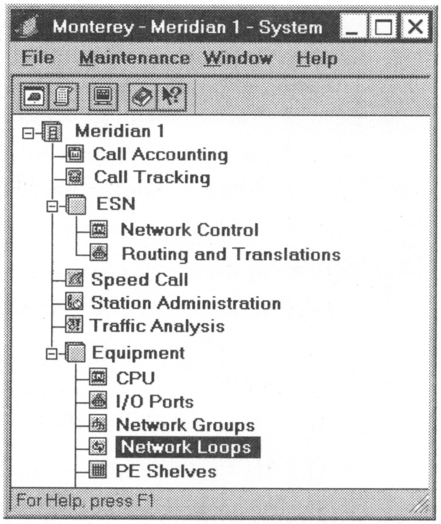
MAT will also normally have direct connections to other equipment: RS-232 to Meridian Mail, and V.24 serial links to MDECT DMC cards inside Access Modules.

7.4.2 The MAT platform

MAT expects to run on a standard Pentium-based PC, with 32 MB of RAM, 1.3 GB of hard disk space, a CD/ROM, an SVGA card, and a 10Base-T Ethernet card. Significantly, this is a relatively normal, commercial platform; our product is the software.

Except on very small systems, the cost of the MAT box is clearly offset by the benefit, and it's just noise in the cost of a big system. Therefore, we were able to make MAT mandatory for Meridian Evolution (but not MSSE), and that in turn let us simplify some of the switch design.

7.4.3 Meridian 1 system view



MAT provides an integrated application to monitor, manage, configure, and maintain M1 systems. This section will not attempt thorough MAT training, but will give an idea of what MAT is like.

Three familiar Windows view types are provided: tree, list, and property sheet. The example at right shows most of the GUI features you'd expect from a Windows application. You can drag, resize, scroll, and hide the view in all the normal ways. Most administrative tasks pop up mouse-navigable dialog boxes with suitable defaults. Dependency failures get propagated through the tree structure. It's fairly obvious that an untrained user would learn his way around this system *much* faster than anyone would ever figure out Overlays, even with the NTPs.

7.4.4 System administration

MAT performs numerous system management tasks. Refer to the User Guide for your software vintage for specific details.

System software installation/archive	MAT Equipment Management application
System configuration	MAT Equipment Management application, CLI Overlays, MAT ESN application
Customer-level data	MAT Station Administration, MAT Overlay Passthrough
Terminal adds, moves, and changes	MAT Station Administration, Overlays
System maintenance	MAT Equipment Management application, CLI, Overlays
Alarms	MAT Alarm Management application
Call analysis	MAT CDR, MAT Call Tracking
Traffic analysis	MAT Traffic Analysis

7.4.5 Alarm management

MAT provides a standard hierarchy of terms for evaluating and discussing the severity of alarms. There is an Event Default Table (EDT) which maps all legacy events to one of these severity levels, which may be overridden using the Event

Preference Table (EPT). The EPT also allows the craftsperson to specify that the severity level be raised based on the frequency of an event type. Critical alarms will typically also be signaled by a siren or flashing light. The available severity levels are:

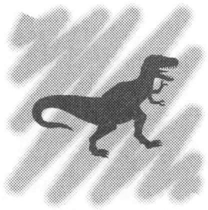
- Critical – things are real bad
- Major – an important subsystem is unavailable
- Minor – something like a pool of resources running low
- Warning – you can ignore this for a while

MAT reports current alarm conditions sorted by severity, event type, time of occurrence, status, etc., and tries to provide advice on how to clear them.

The event log is stored in a circular buffer on the Call Processor, and is preserved across reboots and restarts. It can be viewed through Overlay 117, or with the MAT Event Monitor tool.

The M1 uses SNMP to signal alarm conditions to MAT.

Monitoring	Event Monitor
Persistence	Event Log
Summary	Alarm Banner
Alarm tailoring	Event default and preference tables
Command line passthrough	Overlay 117

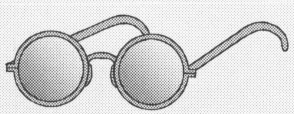


8. Meridian Evolution

8.1 Isn't ME extinct?

Of course not! Or at least the Meridian 1 product line is still leading the market, and will continue to evolve to meet market demands. Now admittedly, Cybele, the project that would have put an ATM switching fabric into the M1, and Pangaea, the project that was building our new line of terminals, were unceremoniously decapitated. And unhappily they had come to be known as "Meridian Evolution". Therefore, the evolutionary path is temporarily hard to find, but we might reasonably expect to see a new plan very soon, perhaps one liberally peppered with terms like "IP" and "webtone". And it seems entirely likely that this plan will call for substantial changes to both the architecture of the M1, and the market it serves, and that to get there from here various people will try to rescue much of the work that was done under the ME banner.

What follows is a Cybele time capsule. It hasn't been changed since before the project was cancelled, and it probably will be removed in the next edition of this book. I've left it in to record for posterity the substantial amount of work done, to allow people to understand some of the issues that would have had to be addressed for it to have succeeded, and to show that it contained some good ideas that the salvage crews should try to extract if possible.



The vision thing

How should the M1 evolve to keep pace with the times? Given the 20th century's dramatic, sustained rate of technology mutation, how can we build anything that will be marketable for more than a few months? The guiding vision that attempts to address these fears goes something like this. We will build:

- person-to-person communications
- regardless of location
- over any media
- using any device
- simply and easily
- securely and reliably
- easily controlled by computers

It's a pretty good start. You can see in this outline most of the orthodox Nortel values, and yet it's hard to argue against it as the system you'd like to have at the "end" of the evolution process.

This chapter will describe what we've done recently to move towards this vision. Bear in mind that the "M1 Rules" already described in the rest of the book will mostly still apply, both because the domain still has many of the same essential properties, and because Meridian Evolution reuses much of the original M1 hardware and software.

Newton's third law of management asserts that for every vision there is an equal and opposite revision; the goal we're currently working towards *will* shift before we get there. Up until this point, the information covered by this book has had a fairly long half-life. Some of it has been true for the past twenty years, and is likely to continue to be true for another twenty. This chapter is inherently less stable stuff. It talks about things that are not on the market, and may not even be built yet. Please forgive me if it looks a bit quaint after a few years.

8.2 Evolving the PBX hardware

When most people say "Meridian Evolution", they're talking about the results of the Cybele project. That is, Meridian 1, but with an Asynchronous Transfer Mode (ATM) switching fabric. In its first incarnation, ME has all of the usual merits of M1, plus the following new advantages:

- non-blocking networks
- simpler, more reliable cabling (OC-3), plus an inherent ability to distribute peripherals up to 2 km with multi-mode fiber, or as far as 15 km with single-mode fiber (which will allow some customers to substantially reduce administrative overhead by consolidating multiple existing M1s on a campus with a single new Call Server)
- dramatically simplified network engineering and load balancing
- cheaper (both direct cost and total cost-of-ownership)
- higher capacity (more ports, more connections, and more calls per hour[28])
- "multimedia ready"

That last bullet deserves some explanation. ME is ready for multimedia in the sense that the INS can connect high or variable bandwidth channels. What ME does *not* have is a native-ATM or other high-speed data access peripheral (although we should expect something in the near future). ME Access Modules still only connect your basic 64 kbps Constant Bit Rate (CBR) streams, just like the original SL-1 did 25 years ago. This means that for now ATM to the desktop (already a suspect solution due to cost) is mythical, and the "integrated ATM backbone" requires you to convert the TDM trunks back to ATM (with something like Passport), which you could have done precisely as easily before we had ATM switching.

Speaking of revisions, one of the original goals of ME has already shifted with the changing LAN market. In the time since ME was first considered, TCP/IP over Ethernet has gone from being an important LAN technology to totally dominating the LAN market. During that same period, ATM has moved to command the WAN backbone, but has lost momentum as a desktop technology. Thus, the original gamble that people would use M1 as their primary data backbone switch has become a bit of a long shot. Besides cost, one technical reason for this is that there is no real shortage of bandwidth to the desk. ATM does the most for you when you're trying to mix a large number of payload

[28] Currently a myth, but we should be able to address capacity issues by the time we get to market.

streams with different characteristics. The last hop to the desktop has very little need for multiplexing, and any cheap, high-bandwidth pipe will do. This does not mean that switching and transmission should not and will not use ATM.

Why ATM?

In the early '90s, all the crystal balls were predicting a merge in voice and data transport. To prevent data equipment vendors from walking off with our market, we saw that we would need a plausible story that would convince customers that their "legacy data" could move nicely through our wires. Technically, ATM looked (and still looks) like the best way to do this. Its advantages include:

- mixing variable and constant bit rate traffic
- lowering costs by mixing real-time with non-real-time applications
- supporting guaranteed Quality of Service (QoS), which Ethernet still can't do[29]
- having low overhead/high capacity for a given wire speed
- having proven scalability to multi-gigabit speeds
- being connection-oriented: inherently more efficient and better QoS, but also inherently trickier to handle topology changes mid-call
- being standards-based technology (eg: we *bought* the INS)
- having some successful ATM networks deployed worldwide (although there are orders-of-magnitude more examples of IP networks…)

An added bonus was that, as one of the four founding members of the ATM Forum, Nortel had some ability to make the standard look like we wanted it to.

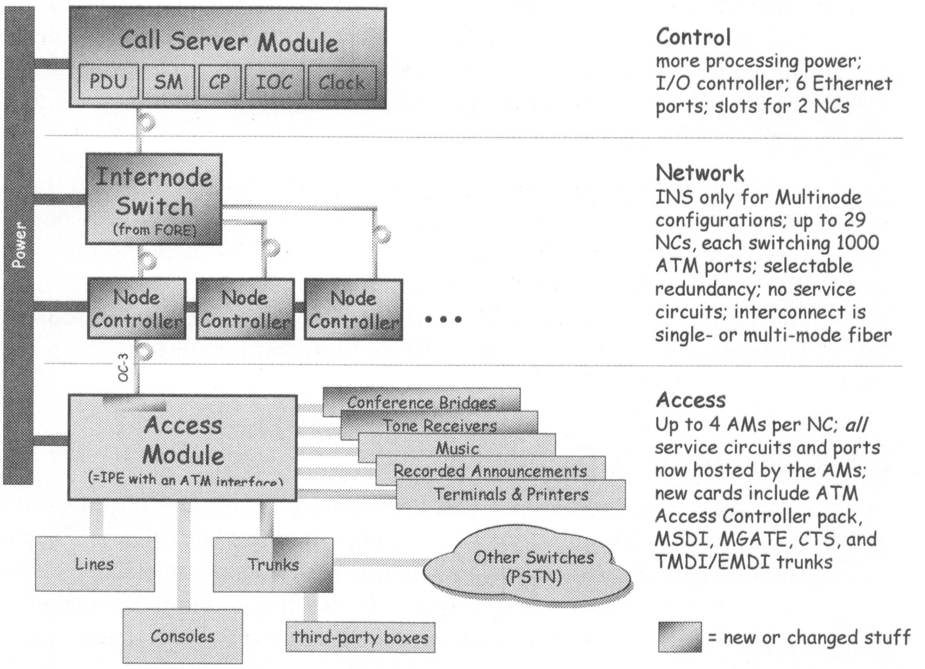
Despite the above arguments, TCP/IP over Ethernet is by far the dominant choice for carrying data to the desktop in today's market. This is partly because it's well-optimized for data and has many reliable vendors, but perhaps mostly because the Network Interface Cards are incredibly cheap, and are starting to become free as vendors include them by default on their motherboards. Assuming that in the very best case we might hope that ATM would displace *a portion* of this market, it seems obvious that a good Ethernet wiring closet solution will be imperative. On-Ramp can no longer become one. It also seems obvious that we will need to develop a better understanding of how ME can take full advantage of, and not be displaced by, tomorrow's LANs.

[29] But this can't last many more months…

8.2.1 The big picture

For all of its differences, the ME hardware has the same layers as traditional M1 hardware.

Meridian Evolution Hardware



Control
more processing power; I/O controller; 6 Ethernet ports; slots for 2 NCs

Network
INS only for Multinode configurations; up to 29 NCs, each switching 1000 ATM ports; selectable redundancy; no service circuits; interconnect is single- or multi-mode fiber

Access
Up to 4 AMs per NC; *all* service circuits and ports now hosted by the AMs; new cards include ATM Access Controller pack, MSDI, MGATE, CTS, and TMDI/EMDI trunks

The three basic PBX layers described in Chapter 1 are still clearly visible, even if all of the big pieces seem to have changed. Given the above diagram, a customer could be forgiven for struggling to find the "evolution" part of ME. It's worth spending a moment talking about what *hasn't* changed. All of the IPE, the wiring through the building and out to desktops, and the terminals, can stay where they are. This is the bulk of the hassle factor of changing an installed PBX, and it should provide enough inertia to retain most customers over the upgrade. But our competitors will still call it a "forklift upgrade". The marketing angle is that at no point during the upgrade is the total system out of service. While strictly true, this may fail to impress some customers because the catch is that at *some* point during the upgrade each peripheral needs to be taken down.

The layering of hardware is a bit stronger than previously. In the early SL-1s, the cabling connecting the control layer to the switching layer was essentially an extension of the main CE bus. The design was tightly coupled, and NE or cabling problems would often show up as CE bus errors. At the same time, many service and access circuits appeared on the NE shelf, because there was no computing power on, and weak messaging out to, the PE shelves.

In the new model, the CS, INS, NCs, and AMs are much more loosely coupled; they are each essentially free-standing computers, each running VxWorks, networked with OC-3 cables. Access and service circuits are now strictly done at the access layer. Architecturally, this has many advantages. It means errors should be easier to isolate. The different layers can evolve more independently than before. And we should get to make "buy vs. build" decisions more often (starting with the INS).

A mostly unimportant disadvantage of the new equipment is that the packaging is not as slick as that of the older M1. The Call Server and Node Controller have three form factors: wall mount, floor mount, and 19" rack mount. The INS must be rack mounted, but has a quite different appearance from the CS and NC. The AMs and PMs use traditional M1 UEM shelves. The result may end up looking a bit haphazard, and may require a larger footprint for some configurations.

8.2.2 Call Server (CS)

This shelf contains the main Call Processor (CP), Input/Output Controller (IOC), System Monitor (SM), Power Distribution Unit (PDU), Clock Controller, and 6 Ethernet ports. The CS shelf also has two slots available for Node Controllers for small configurations. In the first iteration, the main computing engine is the same family of Motorola CPUs used for M1. Later generations should also include an Intel Pentium option. Call Servers are optionally redundant.

Besides some cost savings and improved performance, the real reason for redesigning this equipment was to fulfill the new clocking and I/O requirements. The main job being performed is identical to that of the CP in older M1 vintages. The new IOC hosts a 170 Mbyte PCMCIA hard disk, a serial port, an Ethernet port, and one or two ATM ports. The clock meets Stratum 3 specs (and therefore also Stratum 4)—not a ±2 nanosecond per day Cesium clock, but good enough for Central Office deployment. The SM monitors the ambient temperature and external sensors (like door alarms), drives external alarm systems, and talks to MAT.

8.2.3 Inter Node Switch (INS)

This is a 4×32 port ATM switch, built by Fore Systems. It is not required for single-node systems, and is optionally redundant for large systems.

8.2.4 Node Controller (NC)

This is where the most fundamental hardware change of the Cybele project happened. The Node Controller is a 1000-port[30] ATM switching matrix, and replaces the older TDM switching shelves. It is optionally and selectively redundant. NCs can not co-exist with older network styles, so the upgrade path is not as evolutionary as we usually like to offer.

TDM over ATM: our own private standard

We worry about latency to the point of paranoia.

The model to which we compare voice switching proposals is the old cross-bar machine: a direct, end-to-end copper circuit connecting two parties. By comparison, the data forerunner is the telegraph, which itself is the electronic successor to the pony express: you gave your packet to a clerk, who passed it to a rider, who lept from horse to horse across the continent, and you hoped that a courier would deliver the message at the far end.

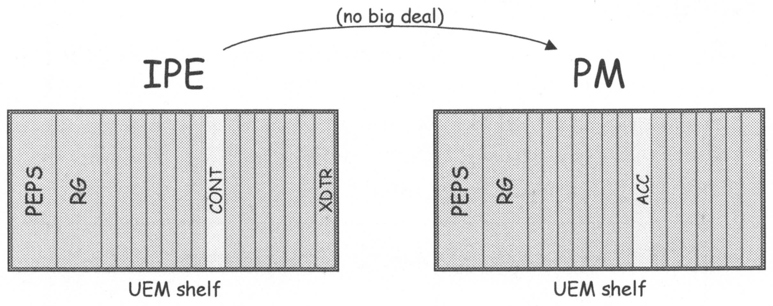
Some voice-over-IP services put hundreds of 0.125-millisecond voice samples into a single IP frame; this improves bandwidth utilization, but delays the voice by several tenths of a second. We were too worried about latency even to fill up a single ATM cell with voice, because it would have added 6 milliseconds (!) to the latency at the initial point of entry into the switching network. So we put each sample from a given voice stream into a different cell, and combine 2 T1s worth of conversations into a single cell, rather than waiting for one stream to fill up the cell. This does reduce latency. It also means we need to do much more work at the start of a call to pre-arrange "time slots" in the ATM cells, leading to a non-standard version of ATM, and we can't simply hook up other ATM equipment to an NC and have it work for free.

And it means that we use *much* more bandwidth that is really necessary, since we reserve a timeslot for each terminal, whether or not it is active.

[30] Actually, it's bigger than this, but we're currently choosing to sell it as a 1000-port, non-blocking switch that doesn't require any engineering effort on setup. Future configurations may leverage this latent power.

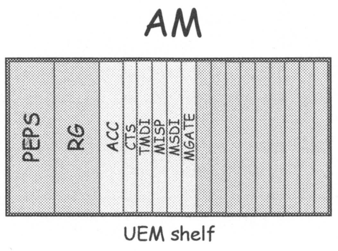
8.2.5 Peripheral Module (PM)

For upgrades of existing equipment, this is really a traditional IPE shelf with a new DS30-to-ATM interface called an Access Controller to allow it to speak ATM to the NC. The Peripheral Equipment Power Supply (PEPS), Ringing Generator (RG), and most other cards can stay where they are.



8.2.6 Access Module (AM)

To host the range of new service cards that replace the older equivalents that used to reside on Network Equipment shelves (ie: CTS, TMDI, EMDI, MSDI, MGATE, and SISP), the current packaging requires customers to configure new AM shelves. AM shelves, still based on the same UEM form factor as the old IPE, are barely distinguishable from PMs. The main difference is that they support the CE-MUX bus, thus allowing a unified trunk card portfolio across the full range of Options.



Earlier in the project, the AM was known as a Peripheral Group Controller (PGC), and the term is still frequently used.

8.2.7 The embedded control LAN (ELAN)

As mentioned above, the boxes just described now form a loosely coupled network connected by OC-3 fibers. This new Embedded LAN (ELAN) allows the components to communicate using industry-standard protocols built on IP, like File Transfer Protocol (FTP), Network File System (NFS), and Point-to-Point Protocol (PPP). The ELAN is used for software download and archiving, system management, application processor messaging, and even call control.

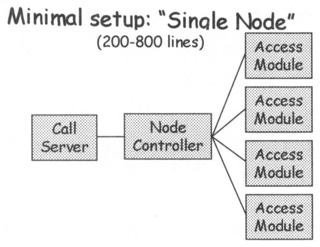
The ELAN comprises multiple physical transports: ATM, call server backplane (IPB), CE-MUX peripheral bus, and DS-0 for mobility EIMCs. While some ELAN messaging uses IP to leverage standard tools, we also do a lot of raw messaging. Our proprietary raw messaging protocol increases performance, understands connection orientedness, supports multiple sessions running over a single virtual circuit, and ties distributed objects together. It is fast but, like IP, it does not guarantee message delivery. (Applications must be designed with this in mind.)

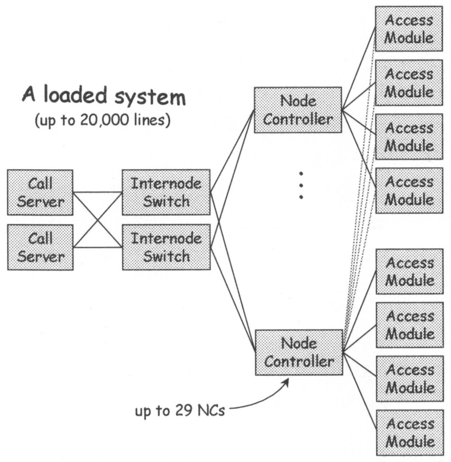
Cybele Message Service organizes the IP/raw messaging transport for ME, delivering messages from one network node to another. It hides the underlying hardware involved, and hence, makes application software more portable. In addition, it provides a mechanism to detect the newly inserted network card.

For a thorough review of IP and raw messaging in the ELAN, see *M01720: Cybele Messaging Service* in Doctool library SL1DOCS.

8.2.8 Available configurations

The ME architecture allows a wide range of configurations. The smallest setup, shown on the left, would usually be wall-mounted, with the NC card in one of the two available slots right in the Call Server shelf. At the top end, we could put together the 20,000-line system shown on the right. Depending on the customer's budget and skittishness, the CS, INS, and soon even NCs can be made redundant.





(The ability to increase redundancy by connecting an AM to multiple NCs is not supported in the first release.)

8.2.9 What about the Option 11C?

The Option 11C hardware is already non-blocking and cost-effective. Therefore, the Meridian Small Systems Evolution (MSSE, not to be pronounced "messy") does not change to an ATM switching fabric. The main advantages of MSSE over Option 11C from a customer's perspective are that MSSE can:

- run ME software (so it won't fall off the development curve!),
- be managed by MAT6 (although unlike ME this is optional),
- drive DTEV sets,
- host ME peripheral cards, and
- run IPE NGen-based applications, specifically SCCS and MCE.

The Option 11C Small System Controller (SSC) has been upgraded with more memory, as well as conference, tone, and I/O resources. The power supply has been beefed up, and a new slightly-pregnant door has been added to allow future faceplate cabling, so the MSSE is compatible with both the new ME circuit packs and most legacy packs. The exceptions are the TDS/DTR, XTD, XMFR, and XMFC/MFE, which have been replaced by the CTS card and the SSC itself.

Comparing costs to benefits, the Option 11C-to-MSSE upgrade may prove to be something of a tough sell. Hopefully, MSSE will at least be attractive to new sites, but if that's the only source of sales, we'll probably have to keep supporting the old equipment for a long time, including new software releases. Since over half of the switches we sell are Option 11Cs we will need to think hard about this part of the business.

8.3 Evolving the PBX software

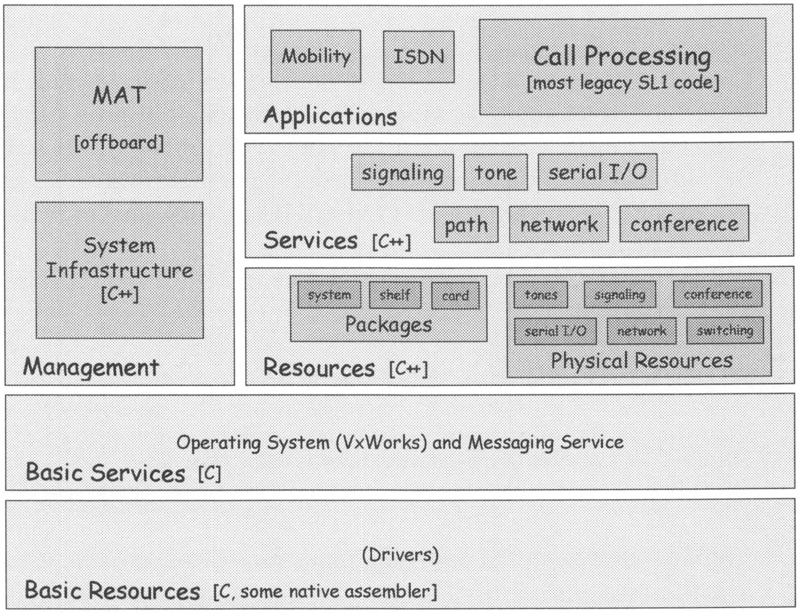
Denny Landaveri's *Software System Architecture* gives a (slightly dated) overview of the ME software. There are also a number of training videotapes available from the IRC, although they are also a bit dated. Be sure to get a copy of the accompanying slides, *Cybele Overview for S/W Designers.*

Although the new ATM switching fabric was the most obvious hardware change in ME, it had little immediate effect on most of the software. After all, most of the code didn't care *how* `SETSPEECHPATH` worked, but only that it accomplished what it was supposed to do. About the only immediate broad impact caused by changing the network equipment was a new TN format (discussed later).

8.3.1 ME software overview

For the obvious reasons, ME endeavored to reuse as much as possible of the original, working, debugged M1 PBX software. At the same time, some basic re-architecting was done in an attempt to make subsequent evolution more painless. This led to the following model. The basic rule here (per the Layers pattern) is that each layer can use only the layer immediately beneath it. The main exception is management, which by its nature likes to prod into all of the other boxes. This diagram is really the top level of the ME class hierarchy, and is best explored by using Rational ROSE to look at the model in `/proj/cybele/model/MEModel`. We will also try to webify it soon.

Meridian Evolution Architecture: Logical View



8.3.2 Objectification

A fundamental tenet in the Cybele creed was that Object Oriented design was good for you. Using OO technology would ensure better designs, more code reuse, and faster time-to-market. C++ designers could be hired off the street, and C++ libraries and tools purchased from third-party vendors.[31]

To make OO designs work in our environment, several strategic restrictions were imposed. The first, known as a "solid state" model, was that object instances should only be allocated once. This avoids both real-time problems and memory

[31] SKEPTIC'S REBUTTAL: Yes, you can hire C++ designers off the street (at a price), but if they apply their usual techniques you may get slow code, memory fragmentation, and worse. While some OO designs are inherently clean, it is easy to write astoundingly bad C++ code. And the hard part of understanding a PBX design is not the language, which has a trivial number of symbols and concepts compared with the millions of lines of code. Code reuse and fast time-to-market require a good process (so people can find, understand, and trust the existing code) and not simply a well-partitioned initial design. We have been unable to buy decent protected memory management or source-code patching tools. And the third-party tools we have found have not been universally bug-free, scalable, or well-supported. But this is not the time to have the C++ debate. It was at least the cool new thing to be working on, and as such directly addressed employee satisfaction, which is especially helpful in an environment where good people have many career choices.

fragmentation common in many standard OO approaches. The second strategy was that as many decisions as possible should be made at compile or loadbuild time. For instance, the mapping of messages to SI queue priorities is table-driven at compile time. These guidelines are useful for precisely the same reasons they always were (as discussed in Chapter 2). Also, classes must be reentrant, because the SI process model allows higher priority tasks to interrupt lower priority ones, rather than preserving the atomicity of transactions the way traditional SL-1 scheduling does.

8.3.3 User Interface concepts

As discussed earlier, the original SL-1 was built with an embedded systems mentality. The first result of this was that the software design was very close to the real hardware, and required changes each of the many times we redesigned circuit packs. The second was that it seemed profligate to spend precious real-time on fancy formatting, or precious memory on help screens. Since then, memory has become very cheap and processing power for a GUI has become available on a separate CPU. With user expectations being much higher than they were in SL-1 days, it seemed imperative to invest in a modern management tool that would help people deal with the complexity of a modern M1. Context-sensitive help added to traditional command line interface (LD 117), many old overlays now obsolete, with functionality rolled into MAT.

The revised MAT tool has a more-uniform state machine (an extension of the ISO model) for managing different kinds of equipment. Equipment status is displayed constantly, so the craftsperson can see abnormal conditions without having to check explicitly. A standard method is available for viewing and modifying individual objects (like cards), container objects (like shelves), and even whole classes (like all ACCs).

The M1 database will no longer be archived on IOP, CMDU, or IODU/C. Instead, the MAT PC can use any commercial removable media, such as floppy disks, Zip disks, or even NFS-mounted devices.

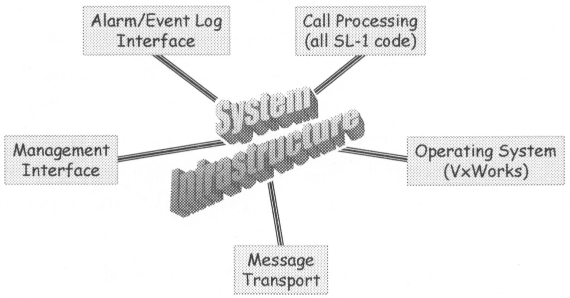
MAT is discussed to some extent in Chapter 7. Many of the changes were in the off-board code, which is outside the scope of this book, but some of them had implications for the switch-based software. The Cybele team created the System Infrastructure (SI) as a foundation for these UI enhancements.

8.3.4 System Infrastructure (SI)

SI is the management framework for ME. It provides policies for operational and management aspects which each component must follow. These policies ensure that each managed object has a standard behavior for serviceability, provisioning, alarm management, and fault detection. The SI object model provides a blueprint for inheritance (for kind-of relationships), composition (for part-of relationships), and connectivity of ME software components. And SI defines appropriate interfaces for industry-standard management tools.

The System Infrastructure (SI), as might be guessed from its name, has somewhat megalomaniacal fantasies of itself as axial to the PBX software.

SI World View


Alarm/Event Log Interface
Call Processing (all SL-1 code)
System Infrastructure
Management Interface
Operating System (VxWorks)
Message Transport

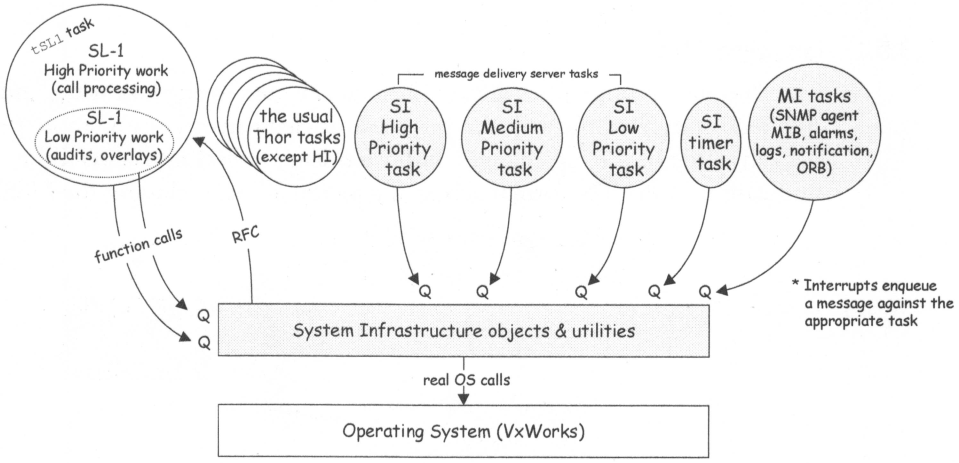
Of course, this is still a more balanced view than that held by `tSL1`, which believes itself to be the *only* piece of software on the switch. While the other boxes in the above picture also communicate with each other directly, and while it is probably correct to consider the main job of a PBX to be call processing and not system management, it is true that SI has a hand in more or less everything that happens on the ME switch today.

SI is the successor to the Hardware Infrastructure (HI) that was built during the Thor project, and is the biggest new piece of software for ME. SI implements what is perhaps the central idea behind ME: the separation of logical (call processing software) and physical (hardware) terminals. SI provides the mapping between these two. SI is a Class Model which provides a blueprint for the design of software components (objects).

Originally envisioned as a tree-structure of hardware components, we found we needed more than that as the system started to get more distributed. The result was a transaction engine with multiple priorities.

The project goals were to provide a software base which:

- is platform-independent
- is extendable
- promotes re-use
- supports the "next generation" PBX (ATM or other networks, message based, managed system, fault tolerant)
- defines interfaces in terms of normalized states and commands
- enforces a design which meets system requirements—there is only one good way to hook into the system, and other attempts will fail entirely



On the core processor, there is also a Management Interface (MI) process which talks to MAT, and of course the SL-1 task, which continues to handle most call processing.

8.3.4.1 Internode communication

SI and VxWorks also now run in many non-core processors. One aspect of the SI's job is to organize a framework for communication between these nodes. Originally, it was thought that Simple Network Management Protocol (SNMP) would be a good solution, but it was found to have performance problems and required a substantial investment in design effort. SNMP *is* used, but so far mostly for autonomous notifications, like fault reporting. Other mechanisms are also in place, most notably the Object Request Broker (ORB) that was inherited by some earlier work on mobility, which is used for request-response dialogues, and NFS, for software download.

Among the basic services provided by SI, which developers can use to build their own features, are a messaging service, a name service, and an event notifier.

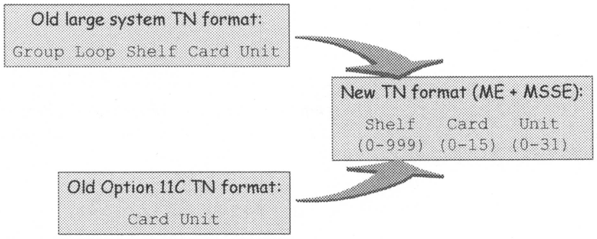
8.3.4.2 Timers

SI provides a new timing service that allows code to run at a certain time of day, or after a certain fixed delay. It is unsuitable for very short (millisecond) delays.

8.3.5 New TN formats

8.3.5.1 Hardware TNs

The Terminal Number has always been closely tied to the hardware, and as such has been contorted through the years as our electronics became more densely packed. One of the big changes with Meridian Evolution has been the unification of the TN format across the product line, including the MSSE.

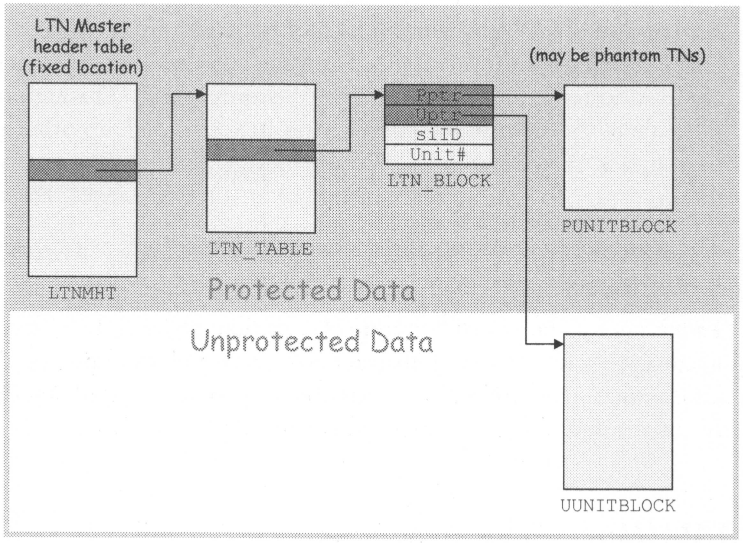

Old large system TN format:
Group Loop Shelf Card Unit
New TN format (ME + MSSE):
Shelf (0-999)
Card (0-15)
Unit (0-31)
Old Option 11C TN format:
Card Unit

8.3.5.2 Logical TNs

Inside the core, there is a new terminal abstraction called the Logical Terminal Number (LTN). LTNs range from 1 to 64k, and should now be used in place of the old TNs throughout the SL-1 code. The Hardware TN format (above) is used only for user interaction in overlay code. An SI identifier (`SIID`) is used to tell SI proxy objects about the TN.

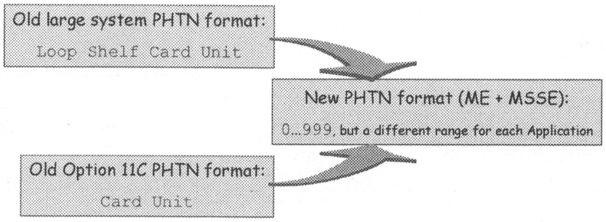
`TNTRANS` will now fetch an LTN item pointer, rather than the old TN item pointer. `LTN_SET_PTRS` may be used before invoking overlay code. `LTNGET_SIID_UNIT` will get the SI identifier to send to SI code.

The LTN Tree



8.3.5.3 Phantom TNs

For Meridian Evolution, the format for each of the above types of Phantom TN has changed to be a simple member number {000 to 999}, and the hardware requirement has been removed. Each of these TN types has a different address space, so there can be a 500-type PHTN 42 *and* a BCS-type PHTN 42. Because PHTNs don't correspond to physical hardware, the `SIID` is set to `NIL`.


Old large system PHTN format:
Loop Shelf Card Unit
New PHTN format (ME + MSSE):
0...999, but a different range for each Application
Old Option 11C PHTN format:
Card Unit

8.3.6 Services

The new ME services attempt to provide a layer of abstraction between the call processing code and the physical hardware. We should now be able to develop new generations of hardware without reworking too much of the existing SL-1 code.

8.3.7 Resources

A managed object on a remote ELAN node requires a "proxy" object in the ME core. Proxies inherit properties from the corresponding physical resources, and are used to track the state of their real counterparts. Through multiple inheritance from "mix-in" classes, they may also pick up other properties like persistence, membership in a redundancy group, or network node status. Proxies then message to remote real objects, or in some instances directly to drivers.

The resource layer of the ME software architecture is comprised of the proxies for the physical resources.

Proxies are instances of concrete classes (like `Digital_Line_Card_Proxy`), which in turn are refinements of abstract classes (in this example, `PE_Card_Proxy`). This inheritance structure makes it possible to provide a lot of standard management functions and default behavior automatically.

8.3.8 RAN/Music/IVR broadcast

Recorded Announcements (RAN), music, and Integrated Voice Response (IVR) systems frequently end up playing the same thing to many listeners at once. A recent enhancement to the M1 software permits a single source to be broadcast to many listeners at once, dramatically reducing the number of packs needed to support a given traffic level. While not strictly part of ME, it is being rolled out at the same time.

8.3.9 New tools

New tools for ME are summarized at http://47.82.33.147/~eddyg/CybeleTools/CybeleTools.html. There's also a largish document called *ME_Tech_Notes* from MLV that talks at a very detailed level about ME and its tools. The most current version should be at http://47.49.4.28:8080/Department/D225/Documents/Useful_Hints/.

8.3.9.1 SIDebug

SIDebug is a collection of C++ macros and class definitions that allow you to add easy debugging support to your code. SIDebug is intended to use as little real time as possible, with the least used when debugging output is not desired. The SIDebug facilities will be shipped with the product, allowing some debugging facilities in the field. It is sort of akin to SNAP for the M1 code. The preceding web link to Eddy's tools page contains further documentation.

8.3.9.2 Technical Support Interface (TSI)

This is the evolution of the Problem Determination Toolkit (PDT). It is designed to support local or remote debugging. It uses the VxWorks shell more heavily than its predecessor, and supports multiple users a bit better.

8.3.9.3 Patching

C++ patching had some new challenges for the platform team, but most have been addressed at this point. Check out the C++ patching web page at http://47.82.33.147/~raviyer/patcher.html.

8.3.10 New memory allocation

In the oldest days of SL-1, we had one page of unprotected data and one page of protected data. You got 4k of each, and no negotiation was possible or necessary. As we grew, we evolved towards the model shown in Section 5.3.1.1. The main goal was to allow each system to grow in a way which best matched its individual need, without requiring the craftsperson to go through a complex engineering exercise.

8.3.11 Third-party software

The third-party software packages Seaweed, RogueWave, Orbix, Envoy, and the SNMP Management Information Base (MIB), which were brought into the M1 platform just before Cybele, are still available. Since most Cybele software is written in C++, these libraries are now useful to a broader range of software than before.

RogueWave classes are mostly used as a template library, rather than inheriting from them. This improves the real-time performance, but does create some code bulk issues.

If you plan to introduce more third-party software to the switch, watch for the following potential problems (all of which have been observed before):

- It won't understand protected memory. Reading protected data is obviously okay. However, if you want it to update protected data, you'll either have to unprotect everything before you invoke it (risky) or copy the results later from a temporary buffer into protected memory (may have race conditions, and will slow down your code).

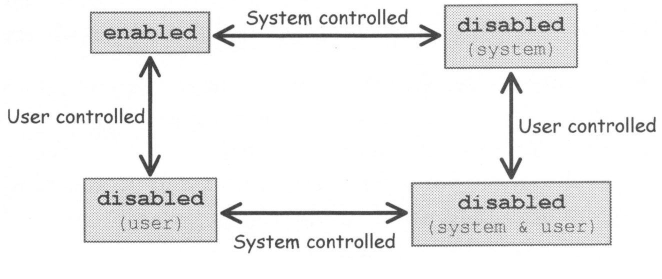
- It may not have been design around the pervasive needs of our system. In particular, a lot of software is poorly optimized for a real-time system that doesn't get restarted regularly. It may make heavy use of `malloc`s, or store things on disk.
- It won't know that registers A5 and D2 are sacrosanct. This may be fine if you're invoking it from C or C++ code, but not from SL-1.
- Behavior under error conditions may be ugly. Test this to ensure that if it traps, all resources are cleaned up and that task is restarted in a sensible way.
- Because we end up relying on an outside vendor, risk management is more problematic. Make sure we have a clear contract specifying support and pricing when it comes to shipping it. Do extra contingency planning. What if they deliver late? What if they lose their source code? What if they go out of business?

8.4 Evolving the rest of the system

8.4.1 New service packs

Since we no longer have Network Equipment, we can't have NE cards, so we had to reinvent the way we provided conferencing, tones, and ISDN.

The new packs share some nice new properties. **Automatic card-detect** provides automatic discovery of field-replaceable components, including card type, serial number, firmware version, and capabilities (eg: number of ports on a conference bridge). **Express provisioning** allows components that have pre-configured default data to be provisioned automatically. **Automatic in-service** allows the system to be configured to always try to put components in service as soon as they are provisioned. And a new **expanded state model** makes managing the new cards more uniform, as shown below:



LEDs on the cards understand this state model, and always reflect the current condition.

8.4.1.1 Multi-purpose Serial Data Interface (MSDI)

The card provides 4 synchronous serial data ports, mostly for AML/CSL, but also for telnet MAT access as an alternative to Ethernet.

8.4.1.2 Conference, Tone, and I/O Server (CTS)

This multi-use DSP card provides 2 serial I/O ports and either 32-party conference or 32 channels of tone transmission and detection.

8.4.1.3 T1 and E1 Multi-purpose Digital Interface (TMDI or EMDI)

Each card provides a single DTI/PRI interface with integrated D-channel processing.

8.4.1.4 Next Generation Platform (NGen)

This is an auxiliary general-purpose computing platform, with built-in digital signal processing capabilities. It comes either as a PC tower or a two-slot IPE card. It provides the required computing power for Meridian Personal Communications Exchange (MCE, formerly called MPCX), Symposium Call Center Server, and DTEV Personal Directory, and is expected to host other applications in the future.

8.4.1.5 Mail Server Gateway Card (MGATE)

This IPE card provides up to 32 port voice connectivity to Meridian Mail through either traditional Network Loop Interface or DS-30XV.

8.4.1.6 Meridian ISDN Signaling Processor (MISP)

There is a family of cards, of varying capacities, which do protocol support for ISDN.

8.4.2 New load processes

8.4.2.1 Amber

"Amber", the new loadbuild process, uses a standard CD/ROM or Ethernet to deliver complete, non-packaged software loads to site. When this file is loaded onto the ME hard disk, the system automatically checks to see if it is more recent than the version in flash ROM, and if so it burns the newer version into flash.

Activation of new features, and system size increases via Incremental Service Management (ISM), are done using an encrypted keycode system. The Distributor Keycode Application (DKA) is a stand-alone, Windows '95-based program used to retrieve and view keycode contents without having to be physically connected to the target system. The keycode is stored in a file, along with a "wrapper" which contains manufacturing and ordering information, and permits access to certain feature packages and ISM limits. Keycode files are normally downloaded electronically, or delivered on floppy, but can be manually entered in an emergency.

dongle
security cartridge

Amber includes the use of a "dongle", essentially an electronically-readable serial number embedded in something that looks like an oversized watch battery, which must match the number the software expects to see. It provides a measure of protection against software piracy. The dongle replaces the older, larger security cartridge, which used to have to be upgraded with each new addition to a switch's functionality.

MSSE has a similar process, but ships software on PCMCIA ROM cards.

8.4.2.2 Peripheral Software DownLoad (PSDL)

Several of the newer peripheral cards (at least the XNET, XPEC, MSDI, MSDL Downloadable D-channel Handler (DDCH), and MGATE) have two banks of flash ROM for storing their software. The general idea is to load the inactive side with new software, and then cut over to it once it's ready. The design means that we can take our time doing the download, since there is no disruption in service during it. Because the software is stored in flash ROM, it's not subject to inadvertent trampling and survives power failures, which means fewer outages and faster recovery times.

There is also an audit task that runs in the background and checks the current load against whatever is stored on disk. If the current load is found to be out of date, it is automatically updated to reflect the most recent load on disk.

8.4.3 New access

The On-Ramp program should provide two important extensions to our product line: a wiring closet solution, and a high-speed access shelf. Existing plans should deliver the former, but with some modification the latter should also be possible, and would greatly improve our multimedia plausibility.

8.4.4 New terminals

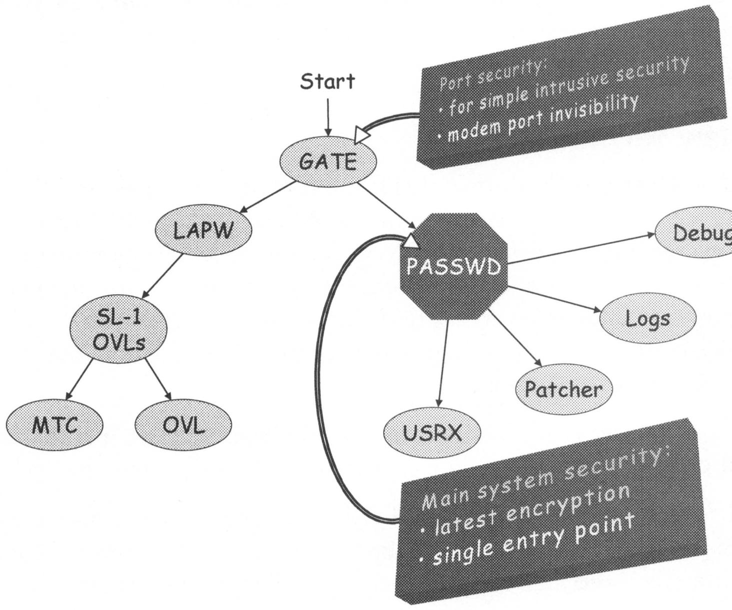
The **Desktop Evolution** (DTEV) program is designed to produce our next generation of telephones. DTEV sets have several context-sensitive soft keys, which may be programmed to activate any of a list of features. One or more "Conspicuous Keys" provides fast access to different application features: personal directory, message waiting, information, feature list, or Computer Telephony Integration.

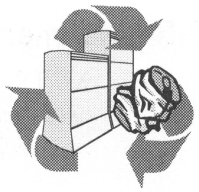
Configuration may be done on an individual set basis, in a template fashion via the DTEV "model" set, or to some extent by the users themselves.

From an architectural point of view, it is worth noting that the new sets are expected to require substantially longer messages (to drive the enhanced user interface), and a directory server somewhere in the system.

8.4.5 New security

ME requires MAT, which may be hooked up to the PBX through the customer's LAN. To cope with the implicitly greater security concerns, a simplified security model has been created. The new model reduces complexity, provides a single point of security, and allows for evolution of services. The customer will still have to ensure that their LAN is configured so as not to pound the PBX with broadcast storms, not so much for security reasons but just to ensure the Call Processor doesn't get distracted from its main task.





8.5 Retiring some older bits

8.5.1 CP

ME requires the new Call Server.

8.5.2 IGS

The old inter-group switch is replaced by the INS.

8.5.3 NE

The older generations of network equipment are replaced by NCs, and can't co-exist.

8.5.4 EPE

There is no evolution plan for the old EPE. IPE shelves may be upgraded to AMs.

8.5.5 SL-1 sets

If there's no EPE, there's nowhere to put SL-1 line cards, so there can't be any SL-1 sets. Since they basically did the job, we may expect existing customers not to be universally delighted that we have manufacturer-discontinued them. Hopefully the DTEV terminals will restore their good humor.

This also meant that we could retire the slow messaging queue, which used to send messages down to SL-1 sets at a pace that would not flood them. We're sort of glad to be rid of the code that did this. It would space out messages using the clock interrupt handler, which meant that a problem sending one of these message led to a bus error *during* the clock interrupt handler. (Remember: do as little as possible during any Interrupt Service Routine!)

8.5.6 Numerous circuit packs

XDTR, CONF/TDS, 3PE, PRI/DTI, MSDL, ESDI, SDI, DCHI, and MISP cards are all going. While they have good replacements, upgrading customers may balk at paying for something they already have.

Appendices

ηιγδ
τη ρεψ

Appendix A: M1 languages

A.1 SL-1

The SL-1 language, and its associated p-code & machine code, are described in detail in Chapter 3 of the Meridian 1 Core course notes. They are available on the web at:

http://47.49.0.148/Department/Training/course_documentation/

The *Global SL-1 Coding Standard* is in the GLOBPROC library of Doctool under G00074.

Of course the best reference for SL-1 is the code itself. Go explore it with x-view. If you'll be doing a lot of SL-1 work, you might want to set your default editor to the xemacs version that MLV and MPK have enhanced specifically for SL-1 viewing, and also look at the sl1-print tool which does pretty-printing of the source code. Both have `man` pages.

SL-1 comprises most of the call-oriented software. It is discussed in detail in Chapter 6.

We are in the process of creating an on-line glossary of commonly-used SL-1 symbols. It should be available by the time this book is published. Try searching the intranet for "SL-1 Glossary".

A.2 C

There are a billion C books around. The classic source is The C Programming Language, Second Edition; Brian W. Kernighan and Dennis M. Ritchie; Prentice-Hall, 1988.

Please follow the *C coding standards for M1*, given under T00045 in the tooldocs library in Doctool. They make the reverse engineering tools usable.

C & C++ code is stored in ClearCase.

C is the language of the operating system, most intrinsics & drivers, and the Hardware Infrastructure. It gives direct access to real pointers (not SL-1 virtual pointers). It lets you multiply by something isn't a power of two. And it's just more comfortable for many newer designers to work in. The code was mostly developed using the GNU C compiler.

Those programmers (like myself) who find that they sometimes need to debug code after they write it are reminded that, although macros seem to be just like procedure calls only faster, it's really tough to set breakpoints in them.

C was first added when we went to a commercial operating system. Since we had to support the C interface anyway to get VxWorks running, people then had the opportunity to program "new" stuff in either language, but in practice this was mostly limited to non-call-oriented software.

From C (or C++), we can get at SL-1 symbols using the macros SL1RAddrOf..., SL1putVar..., SL1getVar..., SL1Rptr..., SL1WSizeOf..., SL1RAddr(), and SL1GetTid(). The *C/SL-1 Interface Programmer's Reference* in Doctool has details. However, we would prefer to minimize the C/SL-1 interface to keep complexity manageable, especially because C and C++ procedures can not be seen from the xview tool. The QSIG and mobility code have done heavy C/SL-1 interactions, and have encountered some challenges along the way.

WARNING

As a real-time optimization, our compilers ensure that register D2 always points to the "context" Call Register and A5 always points to the SL-1 task data. Subroutine calls do not save or restore the contents of these registers, and the code we compile carefully avoids trampling them. However, 3rd-party code compiled elsewhere (specifically, Seaweed, Orbix, Epilogue Envoy SNMP, and Retix/SNACC ASN.1 code) makes no such arrangement. Therefore, you get two problems with the following call sequence:

```
Third_Party_C_or_C++_code (eg: some VxWorks driver) calls
    Our_C_or_C++_code calls
        Our_SL1_code (eg: SL1callGETCALLREG() )
```

First of all, when you hit the SL-1 code, A5 and D2 will have random values, and the SL-1 code may perform approximately random acts. Secondly, if you load the right values into these registers, the original values will not automatically be restored when you return to the 3rd party stuff, and it in turn will have roughly random side effects. These problems may be subtle, and will be very difficult to debug.

Two macros have been provided to help designers to resolve this issue. Calling `SL1saveREGS` from your C or C++ procedures before invoking any SL-1 routines will save the previous values of A5 and D2 and initialize A5 as the `SL1MemoryBase`. Unfortunately, there is no way to determine which call you're working on automatically, so D2 remains as it was. Upon return from the SL-1 code, you can call `SL1restREGS` to restore A5 and D2 to their previous values.

A.3 C++

C++ books also abound. Try *The C++ Programming Language*, Second Edition; Bjarne Stroustrup; Addison-Wesley Publishing Co., 1991, reprinted with corrections June 1994; ISBN 0-201-53992-6.

Some reasonably useful the C++ usage tips are given in the *Mission Park C++ Coding Standard* under G00058 in the GLOBPROC library of Doctool. It is also summarized at http://47.82.33.147/~lang/Process/Standards/html/C++CodingStandard_PROC_app.html.

Not all C++ code is patchable! For the latest guidelines in making your code field-fixable, see http://47.82.33.147/~raviyer/patcher.html.

Once C was up and running, it was only a matter of time before C++ showed up. Newer stuff, particularly Mobility, Cybele, and the System Infrastructure, is mostly written in C++. RogueWave provides a standard class library of useful spare C++ parts.

The code was mostly developed using CenterLine C++, but is about to be ported to the GNU compiler. We had to add a bit to handle protected memory. Essentially, each Object is implemented as two chunks of memory; the protected part contains a pointer to the unprotected part which is updated on warm starts.

If you're new to object oriented design, then besides learning the syntax of C++, you're also going to have to learn about the OO paradigm more generally. You'll also probably want to learn about the Rational Rose modeling tool. Several projects are experimenting with automatic generation of at least some header files directly from Rose models.

While the *Thor Coding Standard* in Doctool recommends including debug code in your source and using `#ifdef DEBUG` to turn it on and off, the availability of source-level debugging tools should now obviate this need. You should still use `rptReport` and `rptTrace` to indicate that some debugging is required.

Header files

The following system header files are available for use. To view one of these files simply establish a Thor context and use the command "ff -d name.h". The path to the file will be displayed and you may now view the file, for instance using "more". To change any of these files, use the PLS update mechanism you would use for any file.

`mcsTypes.h`

This files contains types for the C code written for Multimedia Communication Systems products. It contains generic defines such as ERROR, INT32, TRUE/FALSE, etc.

`mcsMacros.h`

This files contains macros for the C code written for Meridian Communication Systems products. It contains, for example, macros to access 32, 16 and 8 bit words.

`SL1Types.h`

Proposed new file for MSL-1 specific types. This file does not currently exist, however it will be created and updated as needed.

`SL1Defs.h`

This file contains SL-1 language specific declarations such as SL-1MemoryBase, and the real/virtual macros SL-1RAddr and SL-1VAddr.

`SL1Pool.h`

This file contains all of the macros that are automatically generated from the pool used to access SL-1 data structures from a C program.

`gamma.h`

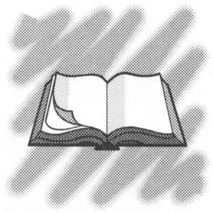
This file contains constants specific to the Option 81 Call Processing (CP) board. It should be included by any module which needs to know about the hardware configuration of the operating system. This file includes intTypes.h.

A.4 Omega assembler

The Omega assembly language, used for SL-1 intrinsics, is described in *SL-1 Omega Assembler Specifications*; D. Landaveri, 22 Aug 1984, BNR Mountain View, Ca.

The Microtec Meta-Assembler was the basis from which the Omega assembly language was built. It is described in *Macro Meta Assembler Manual*; Microtec, Version 6.0, Microtec Sunnyvale, Ca.

Assembly language was occasionally used for firmware, some of the old machine-specific code, and a few intrinsics.



Appendix B: Glossary

This glossary is more-or-less optimized for M1 folks, but there are some other, broader ones available on the web. In particular:

Systems Engineering also provides a nice glossary of terms at http://47.201.134.83:8080/WWW/Administration/Abbrevs/abbr.html

Acronyms Commonly Used in BNR (remember BNR?) are at http://47.80.10.52:5555/docs/public/IT000067_1.0/html/IT000067_1.html

The "Acronym-O-Matic" tool at http://47.4.193.9/~aar/aom/index.html is also okay.

"SHAB": Show Acronyms/Abbreviations has a monster (28k entries!) list of industry acronyms at http://47.202.33.26:8080/~gaikin/shabpage.html

Finally, the Free On-line Dictionary of Computing (FOLDOC), which grew out of the hacker jargon file (http://sagan.earthspace.net/jargon/), is a phenomenal source for understanding computer lexicography, history and culture. There are several mirror sites: try http://www.nightflight.com/foldoc/.

A31	the Integrated Voice & Data signaling chip on the IPE shelf (handles up to 640 sets)
ACC	Access Controller, the CPU in the new Access Module or Peripheral Module
ACD	Automatic Call Distribution
ADAC	Advanced Distributed Access Controller, the next generation of the AM
ADPCM	Adaptive Differential Pulse Code Modulation, a technique for carrying PCM voice with fewer bits, whereby each value sent only gives the difference between the current sample and the previous one
ADPGC	Advanced Distributed Peripheral Group Controller, now called the ADAC
AIN	Advanced Intelligent Networking, a set of public network standards in which call processing services which are controlled by separate, off-switch computers
AM	Access Module (in ME)
AM	Application Module (in M1)
Amber	project name for the new Meridian family order processing/software distribution/configuration strategy, based on CDROM, Internet, and PCMCIA
AML	Application Module Link
ANI	Automatic Number Identification

ANSI	American National Standards Institute
API	Application Programming Interface—a stable, often standardized, method for applications to use a utility, especially across different vendors, versions and platforms
ATM	Asynchronous Transfer Mode—a connection-oriented protocol based on 53-byte cells that allows voice and data to be carried reasonably efficiently on the same carrier
audiotex	premium rate (eg: "900") calls that provide menu-driven information, entertainment or services using IVR technology, sort of an audio-only forerunner of internet surfing
BCS	Business Communications Systems
BCS sets	the SL-1 set + digital sets
bearer	refers to the channels that carry payload voice or data, as opposed to those which carry signaling information (used especially for ISDN)
BHCA	Busy Hour Call Attempts—a unit for measuring switch capacity
BIX	Building Interoffice Crossconnect—a Nortel wiring frame where we map switch ports to a building's telephone cables. It resembles a rat's nest, only with more colors.
BRI	Basic Rate Interface
BSP	Board-Support Package—a collection of VxWorks drivers
BSS	Blocks Started by Symbol—uninitialized store in VxWorks (or Unix)
C	Since Thor, used as a suffix on the Meridian Option names, as in Option 11C, to indicate "Commercial" processor—the MC680x0 family of CPUs...except that both Option 81 and Option 81C use Motorola chips
CardLAN	IPE shelf maintenance communications protocol, between the CP and the peripheral cards
CAS	Channel Associated Signaling
Casper	next generation Attendant Console
CCITT	Consultative Committee on International Telegraphy and Telephony (now the ITU-T)
CCR	Customer Controlled Routing, lets a customer set up a simple script to direct incoming calls through a maze of announcements, menu options, etc.
CCS7	Common Channel Signaling No. 7, also known as SS7
CDB	Customer Data Base, the configuration information on a switch
CDN	Controlled Directory Number
CDR	Call Detail Recording
CE-MUX	Common Equipment Multiplexed bus—the basis for CP control of Network Equipment
CLASS	Custom Local Area Signaling Services—a range of features designed for public telephony, based on the ability to display information like calling line ID using a low-speed in-band modem.

CLEC	Competitive Local Exchange Carrier (as opposed to the entrenched ILECs like Nynex, Pacific Bell, etc.) in the North American dialtone market. For obvious reasons, CLECs are often very aggressive about alternative access strategies built around cable TV, wireless, and TCP/IP. They were sort of invented by the Telecom Act of 1996, and are rapidly evolving to include interstate traffic. Compare RBOC, ILEC, IEC.
CLI	Calling Line Identification (generic term)
CLID	Calling Line Identification (term used in ISDN configuration)
CMA	Changeover Memory Arbitrator—decides which memory is valid
CMB	The later Thor version of CMA
CMDU	Core Multi-Disk Unit (2 SCSI floppies, 1 hard drive)
CNI	Core-to-Network Interface
CO	Central Office
CORBA	Common Object Request Broker Architecture—the OO way for distributed objects to talk to each other
CP	Call Processor—the brains behind the M1 system
CP2	MC68040 Call Processing hardware with new memory management
CP3	MC68060 "
CP4	bus & memory upgrade to CP3, with some software optimizations
CPP	Pentium "
CPE	Customer Premises Equipment
CPLUS	PC-based attendant console
CPSI	Call Processing Serial Interface
CPU	Central Processing Unit
craftsperson	The generic name for the person who runs a switch. In the public switching world, this is a telco employee, but on a small PBX it may be almost anyone. Stereotypically, he will be high-school educated, trained on the job, careful and tidy, somewhat resistant to change, and possibly unionized. Especially in the global market, he may have limited English skills. While individuals obviously vary, if someone with this profile can't run our product, we're in trouble.
CR1	Cybele Release 1—the obsolete name for the obsolete ME Release 1
CSL	Command and Status Link—an interface between the Meridian 1 and Meridian Mail that was the precursor to AML, based on the ISDN Application Protocol
CTAS	Customer Technical Assistance
CTI	Computer Telephony Integration, either off-board call control or simultaneous delivery of "screen pops" of customer data to an agents computer with a customer's call
CTS	Cellular Terminal Server—the controller for all cellular activity on a wireline switch, runs on the EIMC card

CTS	Conference Tone Server pack for ME—has 2 serial ports, plus 30 configurable ports for either conference bridging or tone output. On MSSE, must either be all conf or all tone.
Cybele	Initial project name for *Meridian Evolution*. Cybele was an ancient Phrygian nature goddess ("The Great Mother of the Gods"), whose early worship, involving music, dance, pine trees, and ecstatic self-mutilation, probably led to the Orthodox Christian veneration of Mary. Her consort, ATTis, died violently in various mythic traditions. It's not clear how much she knew about ATM.
DASS2	Digital Access Signaling Services 2—a British private networking standard
David	Not Goliath—internal project name for Option 11 (a small switch, optimized for 96 lines) which was also known as "Fox"
DCH	ISDN D-Channel Handler
Delta	project name for M2317 digital set—`DELTA_2` shows up frequently in the code
dialplan	arrangement of telephone numbers for sets, trunks, and possibly features on a switch
DID	Direct Inward Dialing
DITI	DIgital Trunk Interface—the payload carrier for ISDN services
DKA	Distributor Keycode Application—a Windows '95 program
DMS	Digital Multiplex System—Nortel's main Central Office switching product line
DMS-10	Digital Multiplex Switch-10—a small-line size public carrier switch that originally forked off of the original SL-1 product line.
DMT	Digital Modular Terminals—one of many old names for DTEV
DN	Directory Number
DNIS	Dialed Number Identification Services
DPN	Digital Packet Network—one of the products that originally forked off of the original SL-1 product line.
DRAM	Dynamic Random Access Memory—cheaper but slower than Static RAM
DRU	Dual-mode Radio Unit—the business end of the TRU
DSP	Digital Signal Processing, or Digital Signal Processor
DTEV	Desktop Evolution—the new phones
DTI	Digital Trunk Interface card (non-ISDN T-1 trunks in North America and Japan)
DTI2	Digital Trunk Interface card (non-ISDN E-1 trunks in countries other than North America and Japan)
DTMF	Dual-Tone Multi-Frequency signaling (for "tone" dialing)
DTR	Digital Tone Receiver—an EPE card
E&M	Ear & Mouth trunks—also explicated by some glossaries to mean Earth & Mark, or rEceive & transMit, and even *Excipere & Missere* (allegedly Latin for receive and send)

E1	2.048 Mbps digital trunk—usually carrying 32 × 64-kbps voice channels, the standard in Europe (compare T1)
ECTF	Enterprise Computer Telephony Forum
EIMC	Embedded Intelligent Mobility Controller—the main computing engine for microcellular
EFRC	Enhanced Full Rate Codec—near-wireline voice quality of service for microcellular conforming to IS-641.
EMDI	E1 Multipurpose Digital Interface—a new ME trunk card, includes on-board DCH
EPE	Enhanced Peripheral Equipment (2nd generation)—the predecessor of IPE
ESN	Electronic Switched Network
ETAS	Emergency Technical Assistance Services
ETSI	European Telecommunications Standards Institute
failover	hot swap of a task from a failing processor to another processor
fault tolerance	The ability to continue some degree of operation in the face of component failures (compare *high availability*)
FCAPS	Fault, Configuration, Accounting, Performance, & Security—the more mainstream, ISO-compliant IT word for what telcos call OA&M
FFC	Flexible Feature Code
flash memory	Any of several (≈4) recent cheap, non-volatile, solid-state data storage chip technologies, designed to replace Electrically Erasable Programmable Read Only Memory (EEPROM). Their basic advantage over traditional EEPROM is that they can be cleared and reprogrammed without removing them from the board. This is the memory that survives all restarts, powerdowns, etc. It is not really "Read Only" (or it wouldn't be "Programmable") but it takes an extra jolt of power (hence "Flash") and a bit more time to write to it, so you don't want to use it for volatile stuff. The technology is also being deployed in digital cameras, hand-held computers, cell-phones, etc.
Fox	another internal project name for Option 11
Freya	A Norse goddess of love and beauty, and the Option 21 project codename
FTP	File Transfer Protocol
FX	Foreign Exchange
Gamma	internal name for the Thor CPU board family
GUI	Graphical User Interface
HDLC	High-level Data Link Control—an IBM network protocol
HI	Hardware Infrastructure—the maintenance platform for SL-1
high availability	The ability to provide nearly continuous service. Strategies must detect and recover from system faults, and typically include redundant components, hot card swapping, automatic graceful switchover, and uninterruptible power. Lots of things can go wrong: natural disasters, hardware failure, software

	problems, human error, security breach/vandalism, communication failure, scheduled downtime. Each requires a different kind of solution.
HMI	Human Machine Interface
homologation	The group that prepares product for sale into the international marketplace and knows about standards, protocols, testing, etc. In biology, "homology" is the similarity of structures as a result of similar embryonic origin and development, considered strong evidence of common descent.
I/O	Input/Output
ICCM	Integrated Call Centre Manager—became "Symposium Call Center Server"
IEC	Inter Exchange Carrier, lately more often spelled "IXC"—a long-distance provider, eg: AT&T, MCI, Sprint, WorldCom
IEEE	Institute of Electrical and Electronic Engineers—the international professional society for hardware folks
IGS	InterGroup Switch—connects pre-ME network shelves
ILEC	Incumbent Local Exchange Carrier (usually an RBOC)—the opposite of a CLEC
in-band	sending signals over the same channel bandwidth as service being provided to a customer (such as DTMF tones over a voice channel)
INS	InterNode Switch, connects up to 29 Node Controllers in ME—built down the street by Fore Systems
in-skins	refers to an auxiliary system (like voicemail) that fits seamlessly inside the main cabinets
IODU	Input/Output Disk Unit
IOP	Input/Output Processor
IP	Internet Protocol
IPE	Intelligent Peripheral Equipment (3rd generation)
ISDLC	Integrated Services Digital Line Card—an EPE card
ISDN	Integrated Services Digital Network—a set of public network standards that people had hoped would carry voice & data around the PSTN (prior to the explosive growth of the internet). It's still important for voice traffic, along with country-specific variants like German 1TR6 and French VNR4.
ISL	ISDN Signaling Link, the D Channel for PRI/BRI, or just a stand-alone signaling path
ISM	Incremental Software Management—a scheme for selling software by the port
ITU-T	The telecom standardization sector of International Telecommunications Union, known until March of 1993 as the Consultative Committee on International Telegraphy and Telephony (CCITT)
IVR	Integrated Voice Response
kbps	kilo-bits per second (1000, not 1024)
LAN	Local Area Network
LAPB	Link Access Procedure-Balanced, an HDLC variant used to support X.25

latency	the big issue in real-time software—the time between the occurrence of an event and our response to that event. In real-time software, the maximum latency is engineerable. That is, you can promise that for a given type of event, the latency will be no more than a certain period of time. A late answer may be worse than a wrong answer!
LDAP	Lightweight Directory Access Protocol
Liberator	MCS product for large networks (150-15000 users)
MAC	Medium Access Control—the lower sublayer of the ISO data link layer. The interface between a node's Logical Link Control and the network's physical layer. The MAC differs for various physical media. The MAC address is the hardware address of a device connected to a shared network medium.
MAT	Meridian Administration Tools, the software on the off-board box (SMP) that does OA&M for Meridian 1 systems
Mbps	mega-bits per second (1,000,000, not 1,048,576)
MCDN	Meridian Customer Defined Network
MCE	Meridian Communications Exchange—the successor to "Meridian Mail", a unified messaging platform complete with control by speech recognition
MCS	Multimedia Communications Systems
ME	Meridian Evolution—the "new" name for the cancelled Cybele project
Media-Pro	trade name for existing Meridian software distribution product—the "Software Factory"
Meridian Link	links PBX telephony with host applications, using application layer messages on top of either the LAPB, X.25 or TCP/IP protocols. This is the foundation for Computer Telephony Integration on the M1.
MDECT	Meridian Digital Enhanced Cordless Telecommunications. DECT is a European digital cordless air access standard.
MIB	Management Information Base—lives on the INS & stores all configuration data, which operations are permitted, and any alarms, managed via SNMP get/set and traps
Mirv	an older 16-bit CPU
MISP	Multi-purpose ISDN Signaling Processor card
MMCS	Multimedia Carrier Switch—a strategy to deploy M1s as access vehicles for DMS-250, providing a cheap country point of presence for European alternate carriers.
MNM	Meridian 1 Network Management—SNMP-based, runs on Cabletron's Spectrum
Monarch	MCS product for small networks (5-200 users)
MPCx	Meridian Personal Communications eXchange—old name for MCE
MSDI	Multipurpose Serial Data Interface—the ME version of MSDL, has 4 synchronous RS-232 ports for either ISL DCH or AML
MSDL	Multipurpose Serial Data Link card (lives in the Network)

MSI	Mass Storage Interface—provides the interface between the CPU and the Multi Disk Unit, and holds a security cartridge which must match with the floppy software
MSP	Mobility Service Provider (the EIMC)—the switch interface software in the M1 CPU that the MXC talks to.
MSSE	Meridian Small Systems Evolution (Cybelized Option 11C), sometimes called Half-Node
MTBF	Mean Time Between Failures
MUD	Multi-User Dungeon—a user interface concept begun by Dungeons & Dragons aficionados, whereby users are represented in a space by "avatars", and can see and interact with each other and their environment—sort of a low-tech virtual reality
MXC	Microcellular Transcoder Card—channels voice to and from microcellular radio units
NC	Node Controller—a non-blocking ATM switch that connects up to 4 Access Modules or Peripheral Modules, with 800 ports each
NCOS	Network Class of Service
NEBS	Bellcore Network Equipment Building Specification (GR-63-CORE & GR-1089-CORE)—the physical/electrical requirements for putting equipment into a Central Office
NGen	"Next Generation" Computer Telephony Multimedia Server Applications platform—part of the "Blue" in the Rainbow technology evolution program
NGen IPE	low-end NGen solution for use in the peripheral equipment shelves of Meridian 1 C-Series and Cybele switches, based in Intel x86, with DSPs on board
NTP	Northern Telecom Practice—the documentation we ship with our products
OA&M	Operations, Administration, and Maintenance
OC-3	Optical Carrier, level 3—carries 155 Mbps, equivalent to 100 T1 trunks. This is the new glass connection between ME nodes, replacing the bulky, slower cables.
OEM	Other Equipment Manufacturer—stuff we sell or at least work with that's built by somebody else
Omega	an older 24-bit CPU
OO	Object Oriented
ORB	Obect Request Broker
Orbix	3rd-party object request broker—used by Mobility
OS	Operating System
OSI	Open Systems Interconnection
OSS	Operations Support Systems
Pangaea	an older name for DTEV
PBX	Private Branch Exchange

PCI	Peripheral Component Interconnect—a fast (data-bursting 32-bit, 132 MB/sec) self-configuring local bus standard designed for component-to-component connections for PCs supporting multimedia, full-motion video, etc. CompactPCI is a more rugged, higher reliability mechanical packaging of PCI, and is likely to be the standard for telecom boxes in the near future.
PCMCIA	Personal Computer Memory Card International Association—a compact bus interface standard (renamed "PC-Card" in March 1997). We use the term to refer to cards that fit it, that we ship with pre-loaded software on 40 MB Flash Memory for Option 11C. We also use it for packaging DSPs. Sometimes expanded as People Can't Memorize Computer Industry Acronyms.
PCS	Personal Communication Services—telecommunications services that bundle voice communications, numeric and text messaging, voicemail and various other features into one device, service contract and bill.
PDT	Problem Determination Toolkit
PM	Peripheral Module—an IPE shelf with an Access Controller card added
PGC	Peripheral Group Controller—the older name for the ME Access Module controller
POP	Point of Presence
POSIX	Portable Operating System Interface for Unix
POTS	Plain Ordinary Telephone Service
PPP	Point-to-Point Protocol, the successor to SLIP, uses dynamic IP addresses
PRI	ANSI Primary Rate Interface card (23B+D T1 trunks; 1.544 Mbps)
PRI2	ETSI Primary Rate Interface card (30B+D E1 trunks; 2.0 Mbps)
PSDL	Peripheral Software DownLoad
PSTN	Public Switched Telephone Network—also occasionally called Global Switched Telephony Network (GSTN)
PTT	Post Telegraph and Telephone administration
QSIG	ECMA protocol standard for multi-vendor inter-PBX ISDN signaling
R2	Register-type signaling system #2—usually Multi-Frequency Compelled (MFC) but that's a whole other book
RAID	Redundant Array of Independent Disks—an industry-standard fault-tolerant file system with transparent data recovery in the case of disk failure, originally (circa 1988) a UC Berkeley project to make a big fast reliable virtual disk out of a bunch of small cheap floppy disks.
Rainbow	A strategic Nortel architecture program, whose components were known by various color names, and later by more ethereal names like earth, wind, & fire
RAM	Random Access Memory
RAN	Recorded Announcement
RAS	Return Addresses Stack

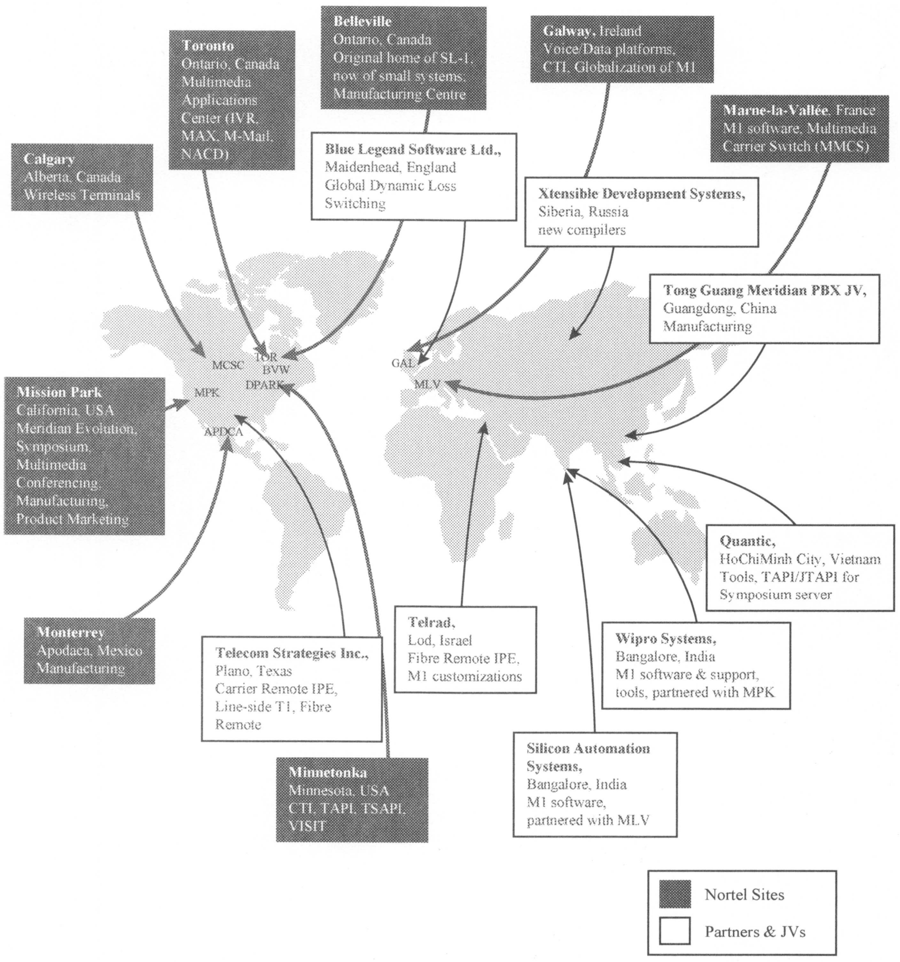
RBOC	Regional Bell Operating Company—one of seven local service providers created by the fission of AT&T in 1984, each of which may in turn comprise several LECs
Retix	3rd-party ASN.1 encoder
RFC	Remote Function Call
RogueWave	a third-party built library of standard C++ functions (time, date, string, linked lists, and some other fundamental structures)
RSM	RS232 Service Module, or maybe Remote Service Module (sources disagree)—a remote management link card for Meridian Mail
SCORE	System Core card—another name for the SSC
SDI	Serial Data Interface
SDRAM	Synchronized DRAM—a newer memory type that is faster because it can be coordinated with the clock signal of the CPU
Seaweed	3rd-party memory management software, brought in by Mobility
SI	System Infrastructure—evolution of HI for Meridian Evolution
SIMM	Single In-line Memory Module—a daughterboard strategy that packages a lot of memory into a small amount of motherboard real estate. Each SIMM card can hold 1, 2, 4, 8, 16, 32, 64, or 128 Mbytes of RAM.
SL-1	Stored Logic 1 (Nortel's first fully-digital switching system); also Switching Language 1
SL-100	a large Nortel PBX, based on DMS-100 technology—ought to be able to handle up to 500k BHCA; the biggest actually deployed today has about 43k lines
SLIP	Serial Line Interface Protocol—allows TCP/IP rlogins to the CPU
SMF	Single Mode Fiber—optical fiber that is very carefully engineered to deliver a signal end to end with exactly one path (no internal reflective paths) so that it can carry that signal over much longer distances than Multi-Mode Fiber (15km versus 3km)
SMP	System Management Platform—now often called MAT (which originally denoted the software running on the SMP, whereas SMP originally denoted the management software running on the M1!)
SMTP	Simple Mail Transfer Protocol—a standard protocol used to drive Meridian Mail
SNA	Systems Network Architecture—IBM's high-level networking protocol for mainframes
SNMP	Simple Network Management Protocol—a network node management protocol, used for a subset of the messages to MAT
SPRE	Special PREfix—the M1 service access codes
SRAM	Static (because unlike DRAM it doesn't need regular refreshing) Random Access Memory, but it will still forget everything if you lose power (unlike Flash ROM), and you can't pack it as tightly as DRAM—usually used for cache

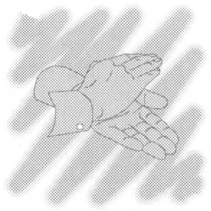
SS7	Signaling System 7—Bellcore term for CCS7
SSC	Small System Controller—providing the CPU, network, and DSP card for Option 11C
SSD	Scan and Signal Disributor—the chip on the old LC machine that detected state changes for up to 4 terminals, built an appropriate 16-bit message, and signaled to the Network Controller terminal scan process that it was ready to send the message
SSERIES	internal name for "small" series of CPUs—the Option 11
ST	Small Turbo—Option 11 predecessor
STS-3c	Synchronous Transport Signal, level 3c, also known as Synchronous Transfer Module, level 1 (STM-1)—usually carried over OC-3 fiber
SWD	Software Watchdog
T1	1.544 Mbps digital trunk, usually carrying 24 × 64-kbps voice channels—the standard in North America (compare E1)
TAPI	Telephony Applications Program Interface—pushed by Microsoft, becoming the *de facto* standard
TCB	Task Control Block (VxWorks)
TCM	Time Compression Multiplexing—an M1-specific way of doing TDM using bit-interleaving and carrying two bearer and two signaling channels per terminal. This is the standard 2-wire interface for our digital ("Aries") sets.
TCP	Transmission Control Protocol—a reliable data connection on top of Internet Protocol
TDB	Trap Data Block
TDM	Time Division Multiplexing—a general technique for putting multiple conversations onto a single wire, typically by interleaving bytes from each conversation (eg: T1)
TDMA	Time Division Multiple Access—a technology that allows a single wire or radio frequency to support multiple calls
TDS	Tone and Digit Switch
Temu	Tape Emulation—we no longer have tape drives, but the file system still has tape emulation because it allowed us to avoid rewriting some code.
TGS	Telephony Gateway Server
Thor	the Norse thunder god; a small town in Indiana (pop. 205); and the original, internal project name to port SL-1 from Omega to MC68k
thoughput	the total amount of transaction processing you can manage in a given time period (cf: latency)
timeslot	a channel in a TDM scheme used to transmit one signal
TMDI	T1 Multipurpose Digital Interface—a new ME trunk card, includes on-board DCH
TN	Terminal Number
TOD	Time of Day

TRU	Transmit Receive Unit—the radio card for microcellular
TSAPI	Telephony Services API—pushed by Novell/Lucent, we support TSAPI on Norstar, M1, and DMS
TTY	historically, a Teletypewriter, although now a generic ASCII terminal
UDP	User Datagram Protocol, a connectionless data protocol on top of IP
UEM	Universal Equipment Module—a standard M1 shelf
UPS	Uninterrupted Power Supply
USB	Universal Serial Bus—a data connector standard
Viking	the project that re-worked the M1 networks and peripherals, creating the "M1" product line and most of the X-things.
VISIT	Visual Interactive Technology—soon to be subsumed into the Symposium label
VME	stands for VERSAmodule Eurocard (only people claim it doesn't because Motorola had already copyrighted VERSAmodule when VME became a standard)—a common small system bus standard
VPIM	Voice Profile for Internet Mail—an Electronic Mail Association standard, based on SMTP and Multipurpose Internet Mail Extension (MIME)
VPN	Virtual Private Network
WAN	Wide Area Network
WATS	Wide Area Telephone Service
XALC	Extended Analog Line Card—an IPE card
XDTR	Digital Tone Receiver—an IPE card
XEM	Extended Ear & Mouth trunk—an IPE card
XMFC/MFE	Multi-frequency Compelled, Multi-frequency Sender-Receiver
XMFR	Multi-frequency Receiver
XPEC	Extended Peripheral Equipment Controller—an IPE card which functionally replaces the old peripheral buffer controller
XTD	Extended Tone Detector
XUT	Extended Universal Trunk—an IPE card

Appendix C: Development sites

For those new to Nortel, it may be interesting to see how work on the Meridian 1 system has become a global development effort.


Belleville
Ontario, Canada
Original home of SL-1, now of small systems, Manufacturing Centre
Galway, Ireland
Voice/Data platforms, CTI, Globalization of M1
Toronto
Ontario, Canada
Multimedia Applications Center (IVR, MAX, M-Mail, NACD)
Marne-la-Vallée, France
M1 software, Multimedia Carrier Switch (MMCS)
Calgary
Alberta, Canada
Wireless Terminals
Blue Legend Software Ltd.,
Maidenhead, England
Global Dynamic Loss Switching
Xtensible Development Systems,
Siberia, Russia
new compilers
Tong Guang Meridian PBX JV,
Guangdong, China
Manufacturing
MCSC
TOR
BVW
MPK
DPARK
APDCA
GAL
MLV
Mission Park
California, USA
Meridian Evolution, Symposium, Multimedia Conferencing, Manufacturing, Product Marketing
Quantic,
HoChiMinh City, Vietnam
Tools, TAPI/JTAPI for Symposium server
Monterrey
Apodaca, Mexico
Manufacturing
Telecom Strategies Inc.,
Plano, Texas
Carrier Remote IPE, Line-side T1, Fibre Remote
Telrad,
Lod, Israel
Fibre Remote IPE, M1 customizations
Wipro Systems,
Bangalore, India
M1 software & support, tools, partnered with MPK
Silicon Automation Systems,
Bangalore, India
M1 software, partnered with MLV
Minnetonka
Minnesota, USA
CTI, TAPI, TSAPI, VISIT
Nortel Sites
Partners & JVs



Appendix D: Credits

The value of this book comes almost entirely from the assembled knowledge of the subject-matter experts in Mission Park, Belleville, and Marne-la-Vallée. The following people all helped make the book what it is:

Charles Babin
Mary Bechtel
Greg Behm
John Boyd
Michel Burger
Dan Calahan
Pascal Cassat
Tej Chaddha
Conway Chan
Stephen Chan
Simon Chiu
Jonathan Crowther
Michael Curry
Tonu Dam
Steven Fraser
Anne Garrison
Eddy Gorsuch
Hukam Gupta
Marjorie Hempstead
Maynie Ho
Greg Hobbs
Ian Hopper
June Hymowech
Roy Isaac
Mason Kudo
Rob la Rivière
Henry Lang
Scott Laribeau
Patrick Masse
Michael McKinney
Pompey Nagra
Dung Nguyen
Andy Phan
Ragu Ragunath
Siamak Razzaghe-Ashrafi
Ann Recktenwald
Ken Roberts
Mary Sipple
Eddie Soliman
Alan Takahashi
Tiffany Truong
Ramakrishnan Venkataraman
Robert Verkroost
Gary Walchli
Sheng Wang
Betty-Anne Wilde
Shawn Wilde
Terri Wood
Lawrence Wong
Doug Zork

If I've missed you, please come yell at me.

The serpent was revealed to inhabit the "Sea of Darkness" to the southwest of Europe by Archbishop Olaus Magnus' *Historia de Gentibus Septentrionalibus* (1555), and arrives in this book courtesy of the Granger Collection. The M1 tank comes to you from the friendly folks at General Dynamics (Land Systems Division). Lanercost Priory (the golden arches) was built in Cumbria by the Augustinian Order in 1220, and debuilt by Henry VIII in 1536. The alchemist's still is from John French's *The Art of Distillation*, circa 1650. The various telco logos in the DAC diagram are owned by Ameritech, MCI, Southwestern Bell, Wiltel, Sprint, and USWest. If you don't understand that Windows is a Microsoft trademark, then you probably didn't understand anything else in the book either. Most of the good M1 diagrams were pirated from various existing Nortel documents.

This book was originally Ian Hopper's idea, and he protected the project long enough to get this edition published.

Karyn cooked, cleaned, proofread, and looked after the Boffin while I wrote.

The whole project owes a secret debt to the work done by the DMS folks to create their *Architecture Thin Guide*, *Programming Model*, and *System Description* books. From any Unix window, the following incantation will produce a good first draft of my book:

```
sed s/DMS/M1/g thinguide pmodel sysdesc > m1guidebook
```

About link rot:

On the day I sent this book to press, all of the web sites I reference worked, ***but*** since the average lifespan of an internet page is only 77 days, you may need to use your favorite search engine to find some of these things. The good news is that, once on the web, information mostly tends to hide, rather than really disappearing.

Name dropping: *Giants mentioned in passing*

Alexander Graham Bell (1847-1922), born in Scotland, but also claimed by the Canadians, invented the telephone (1876) and the telephone company (1877). Also taught the deaf, helped bring photography to *National Geographic*, and experimented with aviation.

Edsger Dijkstra (1930-) is known for his decades of pontifical contributions on how to get algorithms and languages right. He's probably best remembered for his castigations of COBOL, FORTRAN, and BASIC (*GOTO Considered Harmful*); semaphores; and his network algorithms (eg: shortest path, minimum spanning tree), which are regaining importance for things like Open Shortest Path First (OSPF) routing for the Internet.

Tony Hoare (1934-) is Professor and head of the Programming Research Group at Oxford. His 150 papers & books have made many fundamental contributions to the field in the areas of algorithms (eg: Quicksort), languages, OS techniques, methods, notations, proof techniques, communicating sequential processes, distributed computing (the "transputer"). He wrote *Structured Programming* (1972) with Dijkstra and O. J. Dahl. From time to time, he has worked with Nortel.

Alan Kay (1940-) was a founding principal of the Xerox Palo Alto Research Center (PARC), which under his stewardship is credited with having more or less invented everything we recognize in the current desktop computing model:

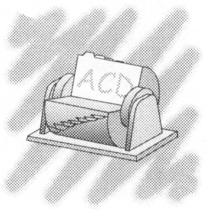
Ethernet, laser printing, the "client-server" paradigm, GUIs with overlapping "windows", Smalltalk, Object-Oriented design, and laptops. His strong interest in education and personal computing then took him to Atari and Apple, and he is now at Disney Imagineering.

Donald Knuth (1938-) is "Professor Emeritus of The Art of Computer Programming" at Stanford University. His 60-odd publications focus on mathematical algorithms, structured programming, and languages. He also invented TeX and coined "Literate Programming", but he is most known for his magnum opus, weighing in at 2200 pages: *The Art of Computer Programming* (1968). Volumes 1-3 are now into a third edition, in at least eight languages. Volume 4 will be out in 2004, and Volume 5 in 2009, but he has no firm dates yet for Volumes 6 & 7 (seriously…).

David Parnas (1941-) is a nomadic professor of CS/EE (so far at UNC Chapel Hill, Technische Hochschule Darmstadt, Carnegie Mellon, Victoria, Queens, Maryland, and now McMaster) who has focused on big, safety-critical, real-time programs, and on how technology can be applied to the benefit of society. He was a strong proponent of data hiding. Besides his academic work (about 180 papers), he has worked on a number of large commercial and government systems, one of which gained him some mainstream notoriety when he resigned from the committee advising the "Star Wars" project, and widely published his concern that the project was fraudulent and dangerous.

Alan Turing (1912-1954) was a British mathematician who made a number of unique contributions to the field. He is mainly remembered for *The Turing machine: On Computable Numbers with an application to the Entscheidungsproblem*, which sort of invented the computer program as we know it; the Bombe, the machine that decrypted the Nazi "Enigma" code; and the "Turing Test", which states that a machine is "intelligent" if you can't tell otherwise by communicating with it. His death by cyanide poisoning, which followed arrest and trial as a homosexual, is thought to have been a suicide.

The **Gang of Four** (named after Mao Zedong's ruling council, or possibly the legendary punk band) are Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. They brought Christopher Alexander's work on patterns into the computing community with their book *Design Patterns* (1995). Buschmann, Meunier, Rohnert, Sommerlad, and Stal followed up in 1996 with *Pattern-Oriented Software Architecture: A System of Patterns*. For more patterns stuff, go look at http://hillside.net/patterns/patterns.html.



Index

Building a better Meridian 1

Some of the M1 code is over 25 years old. That's like, what, three centuries in Internet years? Parnas said a major cause of software aging is "ignorant surgery"—changes made by people who do not understand the original design, which gradually destroy the architecture. Educating our surgeons will be vital if our software is to survive many more operations.

A modern PBX does a complex job. Not surprisingly, this makes the M1 PBX a very complex system. This book attempts to help designers get their heads around this complexity, and understand the programming hazards that lead to outages, broken features, and awkward interaction failures. It aims to reduce the sometimes hidden costs of poor quality, maintenance, and lost opportunities. The insights it is able to impart should directly address the problem of software aging. Its success will mean happier designers, and happier customers. To help readers understand the M1 design, this book separates the key concerns into three main components: context, wisdom, and structure.

Part I: Understanding the context

The first section reviews why our customers buy M1s, the M1 history, and the M1's main hardware components.

Part II: Implications for Design

The second section talks about those pervasive aspects of the design which respond directly to the requirements set out in the first section. The "M1 rules" discussed here tend to apply throughout the software.

Part III: Software Architecture

The third section starts by discussing the overall structure of the M1 software, and then examines each major component of that structure. These chapters should give teams enough working knowledge of the areas in which they are not expert to operate safely within their own spheres.

"Great document – and a fun read."

—Bruce Barrett
founding SL-1 design team member

"Excellent...comprehensive coverage of the art and science of Meridian 1."

—Peter Tarle,
Product Architecture & Advanced Programs, Meridian Small Systems

"Nearly adequate! Please tell me how to improve the next edition."

—Geoff Huenemann,
Author