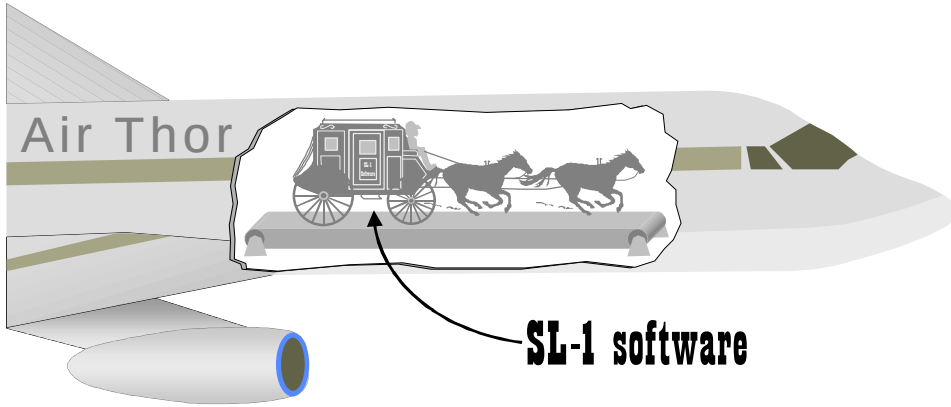
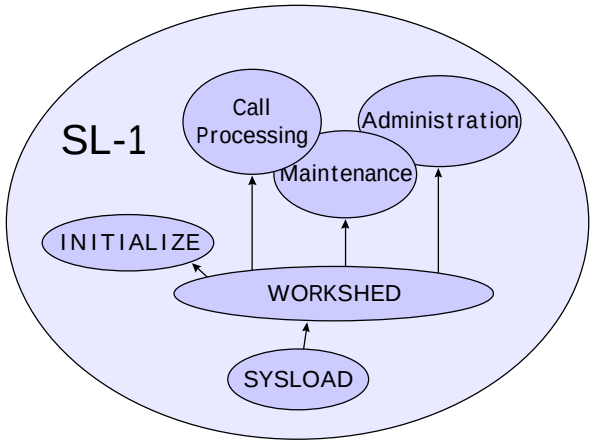
1. The phones (SL-1)

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.)

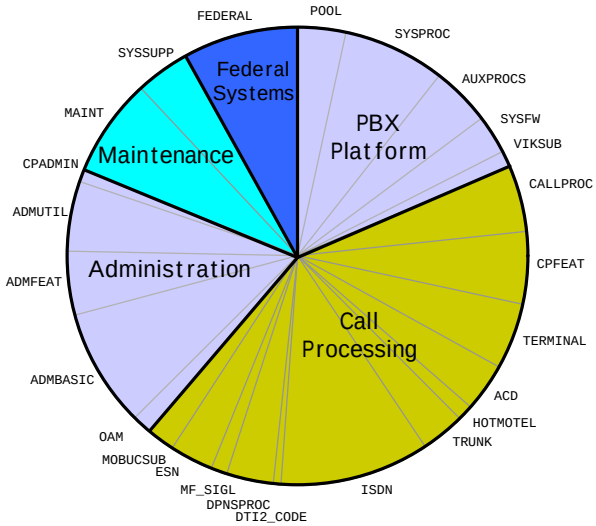
- `INITIALIZE` initializes the call related data structures in the system, and the I/O system, and then returns control to `WORKSHED`.

- 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.

1.1 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.[1]

1.1.1 "Stupid code tricks"[2]

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...
```

---

[1] 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.

[2] 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.

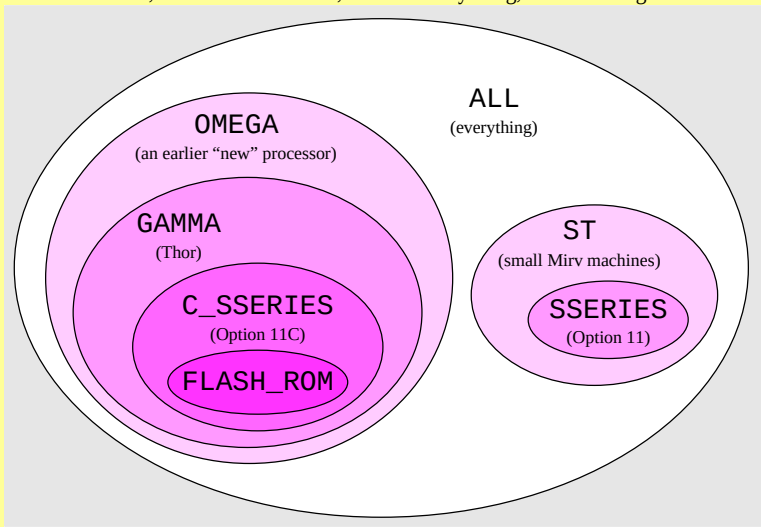
A related but better justified example is the family of constants like "`.BUG4001=4001`". While it appears to be at best a waste of time to have defined them, they do facilitate tracking down the source of error messages. The best place to put these definitions is usually `POOL`, but since it is a major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



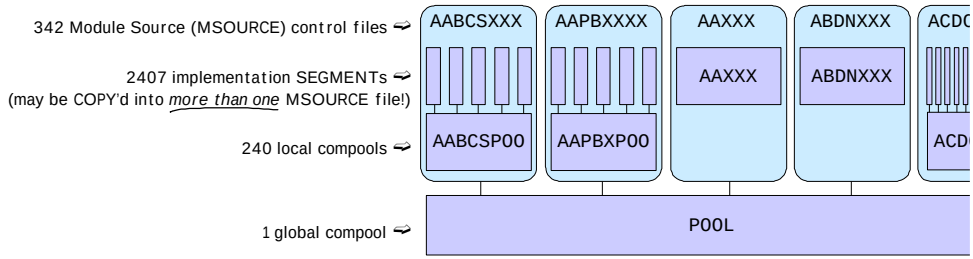
1.1.2 SL-1 code structure

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[3]. 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



[3] 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.

1.1.3 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.

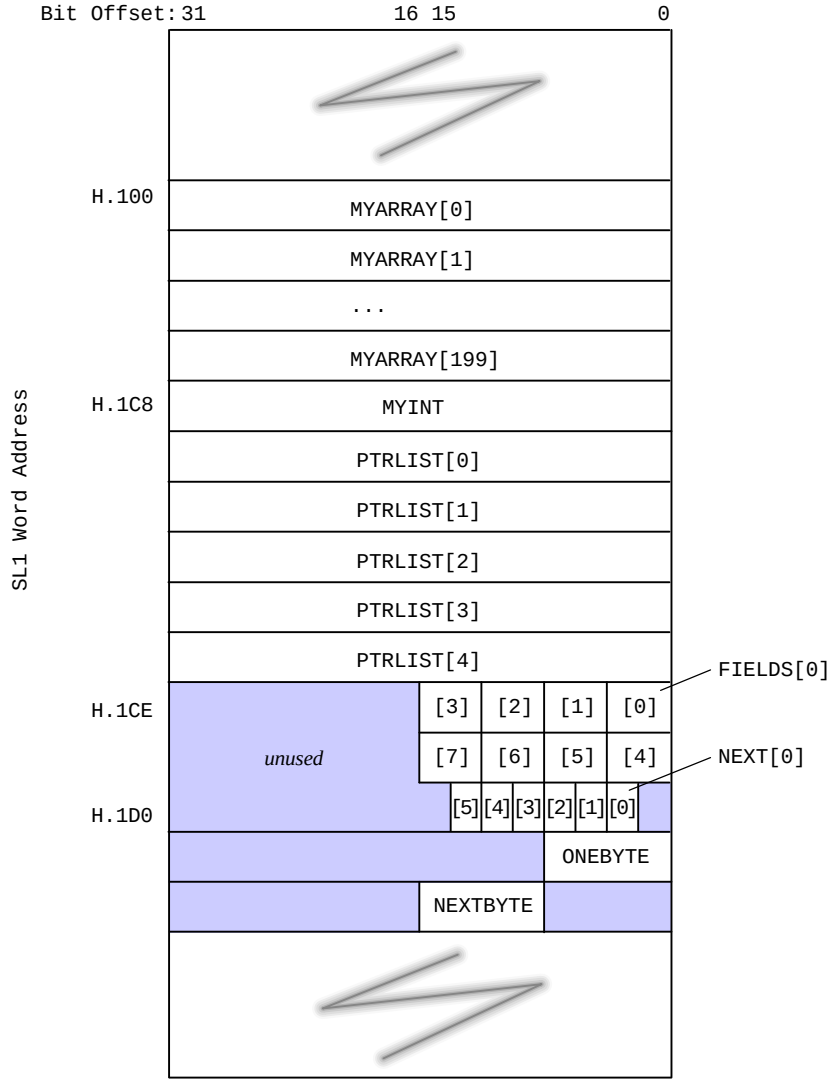
- Packing is big-endian, although we're still puzzling about how best to handle the Intel port. Within the CPU it doesn't matter much, but as was pointed out earlier the M1 machine is really a network of processors, and the messaging can get tricky if the CPU is byte-transsexual.

- 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

1.1.3.1 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.

1.1.3.2 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.

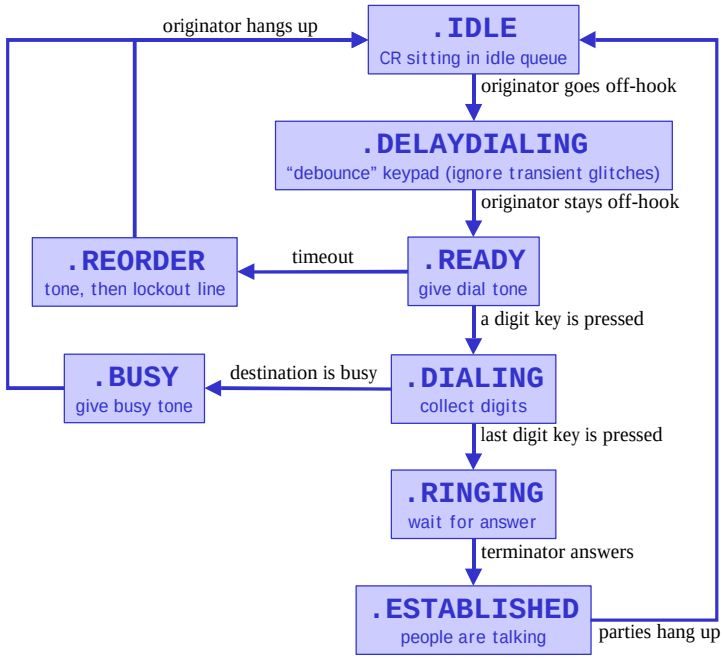
1.1.4 Major SL-1 data structures

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.

1.1.4.1 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[4]), 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.

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

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 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 "`sl1qShow`". 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
```

1.1.4.2 Terminal Numbers (TNs)

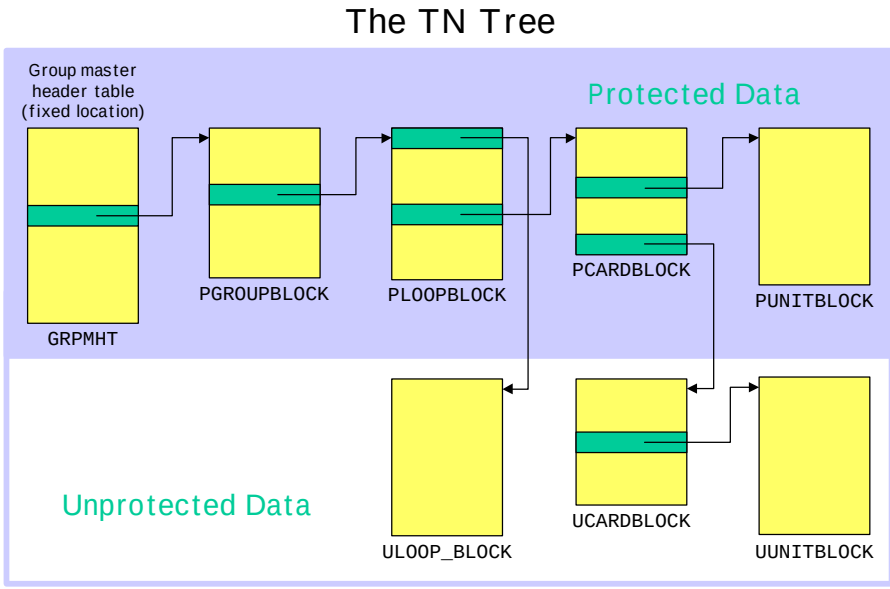
The usual software abstraction for the telephones, trunks, and service circuits involved in a call is the TN. The TN actually specifies the physical location—the network group, loop, shelf, card, and unit—where this terminal's wire terminates on the PBX. Because it is so closely tied to the hardware, 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.



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.

1.1.4.3 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).

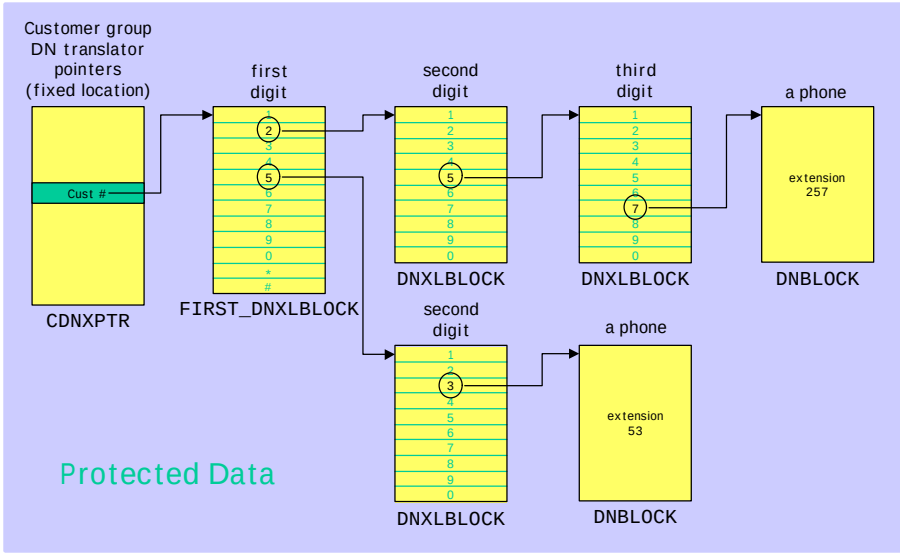
1.1.4.4 Directory Number blocks

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.

1.1.4.5 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.

1.1.5 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.

1.1.5.1 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.

1.1.5.2 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)".

1.1.5.3 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.

1.1.5.4 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.

1.1.5.5 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.

1.2 The PBX platform

1.2.1 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.

1.2.2 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.

1.2.3 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.

1.2.4 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.

1.2.4.1 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.

1.2.4.2 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.

1.2.5 I/O

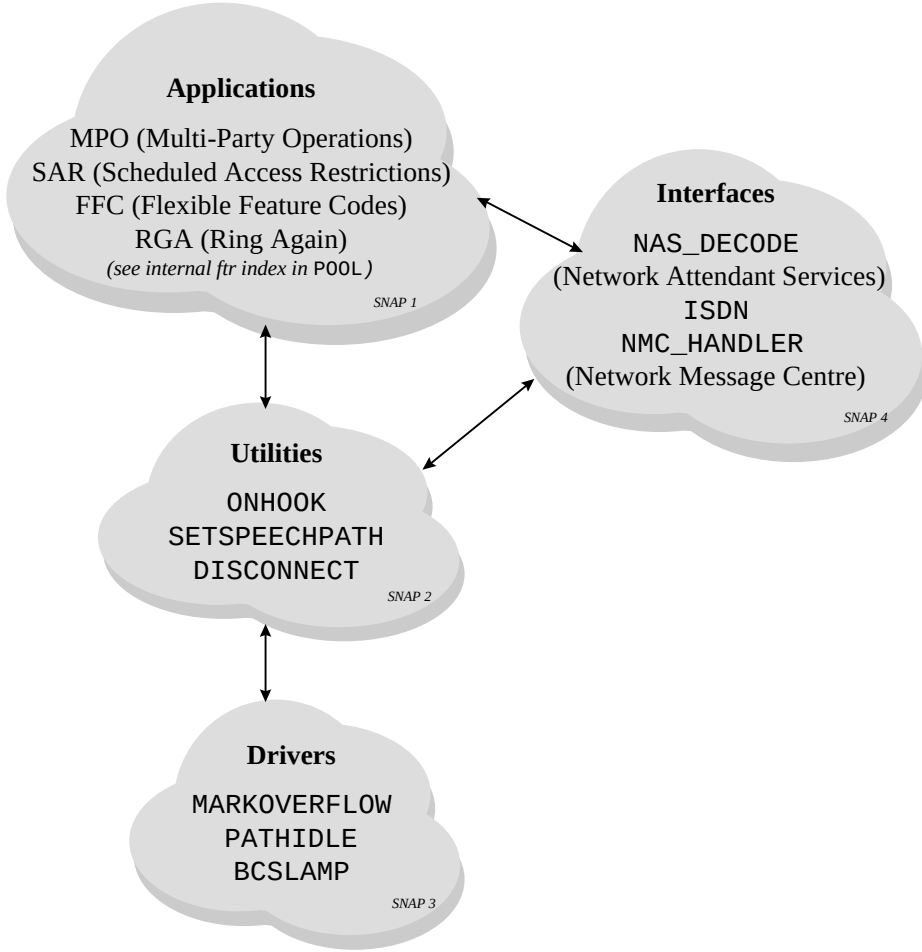
The Meridian 1 has gone through many incarnations. Because each transition was done with limited resources, and because we needed to have a smooth transition strategy for our installed base of switches, large parts of the system were usually left unchanged from one generation to the next. In particular, 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.

1.3 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.]

1.3.1 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!

1.4 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.

1.4.1 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.

1.4.2 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.

1.5 Administration

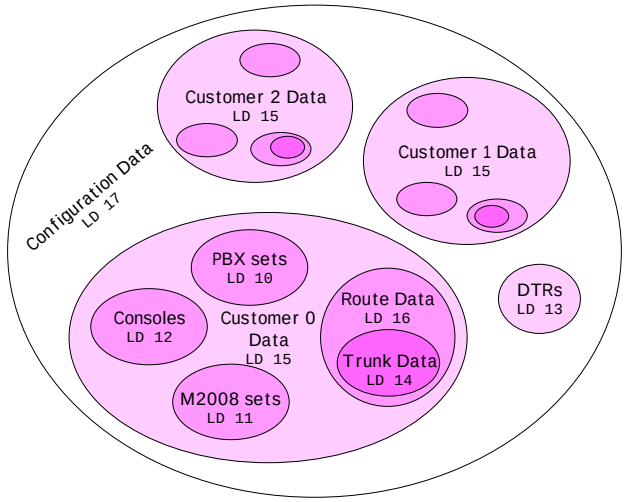
1.5.1 Overlays

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.

1.5.1.1 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.

1.5.2 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.

1.5.3 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.

1.5.4 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.

1.6 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.

1.6.1 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.

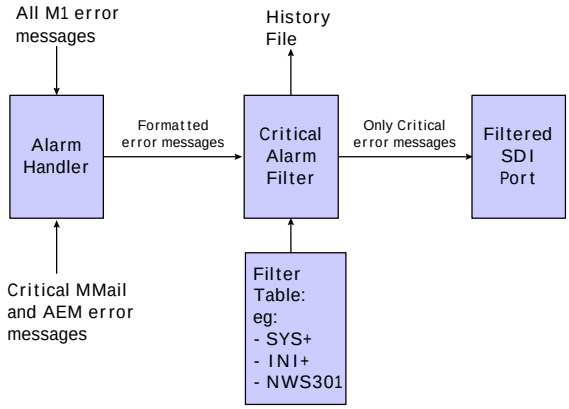
Each error message has:

- 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.

1.6.2 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:



1.6.3 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.

1.6.4 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.