These files list contents of a set of binders I put together for new hw designers here at SCRC. I'm sorry I didn't have time to format them. Where pathnames are given, you can print out your own copies of things.
The gmach files are on white:>gmach>gmach.
The lmach files are on Vallecito:>hardware>boards>lmdp>
lmsq>
lmif>
lmtm>
lmicb>
etc.
Look through the lists and determine what else you'd like. Send mail to me at SCRC or call x7605. There are no block diagrams currently for the IOB, IOP, or memory.

BIRCH @ OSTRICH

```
@heading(Table of Contents)

This binder contains information about Lisp machine architecture of a
general nature.

@begin[description]
How to Get the Information You Need@\

lmarch-directory@\List of files on Vallecito about Lisp-machine
architecture

Excerpts from Training Doc@\Symbolics and system introductions: system
specifications, block diagrams

v-tape.text@\Notes on the video tape series presented by Howard Cannon
[HIC] on the west coast several years ago.  The tapes themselves are
available here at SCRC, but they are time-consuming to view and of poor
technical quality.  Contents include an introduction to some basic Lisp
concepts (data types, EQness, compiled code), a hardware overview (FEP,
DP, SQ), discussions of garbage collection, memory mapping,
microinstruction fields, and the Lbus.

3600.text@\"A loosely organized discussion of various aspects of 3600
architecture"

Architecture of the Symbolics 3600@\A paper submitted to the 12th IEEE
Symposium on Computer Architecture

3600-architecture.text@\

Garbage Collection in a Large Lisp System@\A paper submitted to the 1984
Symposium on Lisp and Functional Programming
@end[description]

Other sources of similar information are the system files,

@begin[itemize]
q:>rel-6>sys>l-sys>opdef.lisp

q:>rel-6>sys>l-sys>sysdef.lisp

q:>rel-6>sys>l-sys>sysdf1.lisp

q:>sys>l-ucode>*.lisp

q:>rel-7>sys>sys2>*.lisp
@end[itemize]

Other information packages available on request are

@begin[itemize]
@I(Symbolics Technical Summary) ← get From Sales Rep

G-Machine Documentation Binder

L-Machine Documentation Binders (Vol. 1 and Vol. 2)

(Customer Service) Maintenance Training Course Student Handout Material ← get from C.S. in Chatsworth

Microcode Manual

LIL Manual (LIL is the FEP programming language.)

NS Manual (NS is a VLSI design-capture system, being expanded to PCB
design.)

(Customer Service) Illustrated Parts List ←―――――――――――― "

(Manufacturing) Component Masters ←―――――――――――――――― "

Product Plans -- see the product manager for the project you need to
know about
@end[itemize]
```

```
@heading(Table of Contents)

This binder contains most of the information currently available
concerning G-machine hardware.  It's contents will be updated and
augmented as new information becomes available.
@I(Caveat:) Note well the dates on the text files.  Some of them are
very old and should be assumed to be out of date.

@B(Section 1:  General Information)

@begin[description]
system-description.text@\Covers briefly early concepts of
system timing, the GBus, and control memory arrangement

doc.text@\A lengthier and newer discussion of above topics,
plus a few others

sys-primer.text@\"A primer to prepare novice system persons for the
reading of the gate-array documents"

register.text@\"Where all the registers and memories live in the NBS"

microinstruction.bits@\The microcode fields and their functions for both
G- and L-machines

backplane.design@\Conventions, pinouts, termination, Lbus signals,
timing, addressing
@end[description]

@B(Section 2: G-machine Processor PCB Prints)

@I[Contact for more information:] Ron Lebel [RJL] or Barton Gawboy [GAWBOY]

@begin[description]
Four processor block diagrams@\

board-layout.text@\

external-grant.text@\LBus differences for the NBS

Timing Diagrams@\

Clock Generator Schmematic@\

signals.xref@\Alphabetical cross-reference listing of all signals on the
G-machine processor board

Listings for the fourteen PALs on the processor board@\

Gmach Processor PCB Schematic Diagrams (26 pp)@\
@end[description]

@B(Section 3: Gate Array Prints)

Text files, xxx-part.text and, for a- and bmema, xxx-doc.text, and prints for
each of the seven G-machine gate-array devices:

@begin[itemize]
amema

bmema

map

ipu

sq

dp

tag
@end[itemize]

@B(Section 4:  FEP/IO and Paddle Boards)

@I[Contact for more information:] Logan Palmer [LPALMER]
```

```
@begin[description]
Two FEP/IO block diagrams@\

fep/io-rev4.spec@\

fep-paths.text@\The Spy bus interface for the G-machine

Gmach FEP/IO PCB Schematic Diagrams (37 pp)@\

Gmach ST506-Paddleboard PCB Schematic Diagrams (13 pp)@\

Gmach SMD-Paddleboard PCB Schematic Diagrams (15 pp)@\
@end[description]

@B(Section 4: SLB Memory)

@I[Contact for more information:] Fred Drenkhahn [FRED]

@begin[description]
spysel.pal@\

Small-Lbus-Board Memory PCB Schematic Diagrams (22 pp)
@end[description]

@B(Section 5: Console)

@I[Contact for more information:] Fred Drenkhahn [FRED] or Logan Palmer
[LPALMER]

console-rev1.spec@\

Gmach Console PCB Schematic Diagrams@\To be supplied
```

```
@heading(Table of Contents)

This binder contains one half of the information currently available for
L-machine hardware. It's contents will be update and augmented as new
information becomes available.  The material in this volume covers the
three processor boards (DP, SQ, and TMC or IFU).  Another binder
contains the other volume, which includes material on the FEP and I/O
boards.

@B(Section 1: General Information)

@begin[description]
Micro-architecture block diagram@\

backplane.design@\

clock.design@\

packaging.design@\
@end[description]

@B(Section 2: L-machine Data Path - DP)

@I[Contact for more information:] Bruce Edwards [BEE] or Dave Moon [MOON].

@begin[description]
data-path.design@\

dp.wss@\

L-machine data-path PCB schematics (38 pages including block diagram)@\

Listings for the nine DP PAL devices@\
@end[description]

@B(Section 3: L-machine Sequencer - SQ)

@I[Contact for more information:] Dave Moon [MOON].

@begin[description]
sequencer.design@\

sq-rev5.wss@\

L-machine microsequencer PCB schematics (50 pages including block diagram)@\

Listings for the six sequencer PAL devices@\
@end[description]

@B(Section 4: L-machine IFU)

@I[Contact for more information:] Chas Horvath [CHAS].

@begin[description]
ifu-upgrade.text@\

theory-of-op.text@\An older, and somewhat misleading, description --
take with a large grain of salt

ifu-control.text@\

x-ref.text@\

L-machine IFU PCB schematics (54 pages including two block diagrams)@\

Listings for the thirty-seven IFU PAL devices@\
@end[description]

@B(Section 5: L-machine Temporary Memory Controller - TMC)

@I[Contact for more information:] Chas Horvath [CHAS].

@begin[description]
tmc.design@\

tmc-rev5.wss@\
L-machine memory controller PCB schematics (38 pages including block diagram)@\

Listings for the twenty-one TMC PAL devices@\
@end[description]
```

```
@heading(Table of Contents)

This binder contains one half of the information currently available for
L-machine hardware. It's contents will be update and augmented as new
information becomes available.  The material in this volume covers the
the FEP and I/O boards.  Another binder contains the other volume, which
includes material on three processor boards (DP, SQ, and TMC or IFU).
@I(Caveat:) Note well the dates on the text files.  Some of them are
very old and should be assumed to be out of date.

@B(Section 1: L-machine Front-End Processor - LFEP-X3)

@i[Contact for more information:] Chas Horvath [CHAS]

NOTE: LFEP-X3 is the latest revision of the FEP board, which was
designed for use in the newer 36xx processors (3640 and 3670).  It
incorporates circuitry that was formerly located on a small PCB, the
nanoFEP (not to be confused with nFEP, which stands for "new FEP," which
pertains to new firmware on the FEP that allows incremental world
saving, among other things.)  Information on the old FEP and the nanoFEP
is contained in this volume, near the end.

@begin[description]
LFEP-X3-address.text@\

LFEP-X3>nanofep.text@\

page-tag.design@\

dma.design@\

LFEP-X3>blk.wlr@\

L-machine LFEP-X3 PCB schematics (24 pages)@\

Listings for the twelve LFEP-X3 PAL devices@\
@end[description]

@B(Section 2: L-machine I/O Board - IOB)

@i[Contact for more information:] Don Tillman [TILLMAN]

@begin[description]
audio.design@\

network.design@\

video.design@\

vd.design@\

vdmem.design@\

net.design@\

vdmctl.design@\

disk.design@\Watch out. This one's very old.

L-machine IOB PCB schematics (45 pages)@\

Listings for the forty-one IOB PAL devices@\
@end[description]

@B(Section 3: L-machine I/O Paddlecard - LMIOP)

@i[Contact for more information:] Don Tillman [TILLMAN]

NOTE: The situation with the I/O paddlecard is a bit complicated.  The
IOP itself (info contained in this section) supports the SMD disk
interface. Other interfaces are supported by two additional versions of
this card:  LMIOP-ST506 and LMIOP-ESDI.

@begin[description]
disk-interface-comparison@\Compares SMD and ST506/ESDI interfaces
```

```
Part of iop.wlr@\

L-machine IOP PCB schematics (23 pages)@\

Listings for the two IOP PAL devices@\
@end[description]

@B(Section 4: L-machine ST-506 I/O Paddlecard - LMIOP-ST506)

@i[Contact for more information:] Don Tillman [TILLMAN]

See the note for the LMIOP.  This set of schematics contains the circuits
on the board specific to the ST506 interface -- the names of these
sheets start with S.  All the other sheets contain circuits common to
both the LMIOP-ST506 and the LMIOP-ESDI.

@begin[description]
disk-paddle.design@\

L-machine IOP-ST506 PCB schematics (15 pages)@\

Listings for the fifteen IOP-ST506 PAL devices@\
@end[description]

@B(Section 5: L-machine ESDI I/O Paddlecard - LMIOP-ESDI)

@i[Contact for more information:] Don Tillman [TILLMAN]

See the notes for the LMIOP and LMIOP-ST506.  The schematics here are
only those sheets that pertain to the ESDI interface.  The other
circuits on the LMIOP-ESDI board are included with those for the ST506
inteface.

See also the file V:>HARDWARE>BOARDS>LMIOP-ESDI>ESDI-FEMPTOCODE.LISP.

@begin[description]
esdi.text@\

L-machine IOP-ESDI PCB schematics (6 pages)@\

Listings for the ten IOP-ESDI PAL devices@\
@end[description]

@B(Section 6: L-machine Enhanced Line Printer PCB)

The ELP interface makes Symbolics line printers look like fancier more
powerful printers.  Part of this interface is on the FEP paddle card.
The part to which the schematics in this section pertain is a board that
plugs into an LGP1.

@i[Contact for more information:] Doug Duncan [DD]

@begin[description]
elp.wlr@\

L-machine ELP PCB schematics (3 pages)@\

Listings for the ELP PAL device@\
@end[description]

@B(Section 7: L-machine 2-MW Memory Board - LM2MW)

@i[Contact for more information:] Marc Hamon [HAMON] or Chas
Horvath [CHAS]

@begin[description]
2mw.wlr@\

L-machine LM2MW PCB schematics (36 pages)@\

Listings for the LM2MW PAL device@\
@end[description]

@B(Section 8: L-machine Front-End Processor - LFEP)

@i[Contact for more information:] Chas Horvath [CHAS] or David
```

```
Plummer [DCP].
See also the text files that accompany the LFEP-X3 section.

@begin[description]
L-machine LFEP PCB schematics (37 pages)@\

Listings for the twenty-seven LFEP PAL devices@\
@end[description]

@B(Section 9: Contents of V:>HARDWARE>*.*)

Hardcopy of FSEDIT for this directory and its subdirectories.
```

Notes on the SMD disk interface, and on building an adaptor to allow
an A machine disk control to attach both Trident and SMD disks
(allowing switch selection of which one is unit 0).

Two cables: "A" (daisy chain) 60-conductor flat twisted pair.
"B" (radial) 26-conductor flat with ground plane.

All signals are driven by 75110A and received by 75108B (or 75107B).

Termination:

The B cable clock and data signals are terminated with 82 ohms to ground
and 470 ohms in series at the receiving end. There is no termination
at the transmitter.

The A cable, and the miscellaneous B cable signals, are terminated with
56 ohms to ground at the transmitter and receiver both. The receiver
also has 470 ohms in series. The Open Cable Detect is driven by two
transmitters in parallel and has no 56 ohm pulldowns.

A cable, controller to drive:

Unit Select 0-3, Unit Select Tag.
Bus 0-9, Bus Tags 1-3.
Open Cable Detect.
Power Sequence Pick/Hold
(these are optional, drive may be put in local mode. If used ground
these with open-collector peripheral drivers. Voltage is TTL level
on most drives, I don't know whether this is guaranteed. 75452
seems to work.)

A cable, drive to controller:

Index, Sector (start of every sector except 0, sector length switch selectable)
Fault, Seek Error, On Cylinder, Ready (On Line), Write Protected
Address Mark Found, Dual Channel Busy (not interesting)
and one spare

B cable, controller to drive:

Write Data, Write Clock

B cable, drive to controller:

Read Data, Read Clock
Servo Clock (available at all times and fed back as Write Clock)
Seek End (level, not pulse), Unit Selected
and two spare

Tags:

The bus must be stable for a minimum of 200 ns before and after any tags
are asserted.

```
Tag 1 = Cylinder Tag   (bits are numbered from right to left as standard)
Tag 2 = Head Tag
Tag 3 = Control Tag:
        0  Write
        1  Read
        2  Servo Offset Plus
        3  Servo Offset Minus
        4  Fault Clear
        5  (Address Mark Enable)
        6  Recalibrate
        7  Data Strobe Early
        8  Data Strobe Late
        9  (Dual Port Release)
```

Differences from Trident relevant to A-machine adaptor:

Trident provides index and sector information for all drives, SMD
provides only for selected drive. (Some SMD drives also provide
index and sector for all drives.) The A machine controller depends
on being able to have a valid sector number; the adaptor could
either allow only one SMD drive and always select it, or force no

sector pulses until the first index pulse after switching units.

Read/Write data clocking is different, but only the adaptor will care I think. Problem comes in switching between read and write clocks which is done while the DC is clocking itself from the bit clock. May have to have a synchronizer here and skip a couple clocks (that's what the Foonly does).

Attention (seek end) cannot be cleared remotely; the controller must provide an edge-triggered latch.

Sequencing and selection don't work the same way. This can be handled.

Must drive Open Cable Detect.

The control signals on the disk bus are completely scrambled; the adaptor will have to look at the tag lines and fix this. The DC microcode already provides a mechanism by which it selects which tag it is going to use (and sends it to the DM board) one microcycle before it asserts TAG ENABLE which puts the tags actually onto the bus. This should provide enough time to shuffle the bus signals around. I believe the microcycles are always 2 microseconds long when using the bus for other than control tag signals.

Also the Trident's Pre Gate should simply be ignored.

The servo offset feature works quite differently; in practice it seems to be worthless so we can probably just not implement it.

The adaptor should receive a Trident disk bus cable as if it was a disk drive, as well as plugging into the edge connector signals for the DM board. It should emanate four disk cables, for one Trident disk and one SMD disk.

Sector Formatting Information (for CDC):

#bytes	Purpose (in order through a sector)
16	Servo-head to data-head tolerance
11	PLO Sync (zeros)
1	Sync Mark (ones)
8	Header and ECC
1	Write Splice
11	PLO Sync (zeros)
1	Sync Mark (ones)
1152	Data (256 36-bit words)
4	ECC
1	Pad (always written)
8	End of record tolerance

1214	Bytes per sector
62	Overhead bytes

Depending on L-machine interface details, an additional gap of a few bytes between the header and data could be necessary to provide for maximum microcode task response delay.

The A machine controller requires longer gaps; see its microcode. Basically it has about a 24-byte delay before turning on read gate for the header PLO Sync and a 10 byte delay over the write splice between header and data. When writing the data field it writes 19 bytes of PLO Sync field and writes 4 guard bytes rather than 1.

Actual sector formatting information for Fujitsu MM2284 on A Machine:

#bytes	Purpose (in order through a sector)
5	Start-block synchronizer delay
16	Servo-head to data-head tolerance
11	PLO Sync (ones)
8	Paranoia (ones)
1	Sync Mark (ones and a zero)
8	Header and ECC
1	Write Splice
13	PLO Sync (ones)

```
7       More sync (Trident compatibility)
1       Sync Mark (ones and a zero)
1       Pad (zeros) (Controller pipelining bug)
1024    Data (256 32-bit words)
4       ECC
1       Pad (always written)
11      End of record tolerance (gap3)
24      Extra delay because of DC inter-sector delay
----
1136    Bytes per sector
112     Overhead bytes
```

18 sectors per track with 32 bytes left over at end of track

This format should work identically on the Fujitsu M2312.

For the Priam, the same sector format should work, except that you only get to control the number of sectors per track, not the number of bytes per sector (goddamned "smart" interfaces). There is room for 17 sectors per track which means 1183 bytes per sector. The format program wants things to be word-aligned, so it will write 1184-byte long sectors. There are 49 spare bytes at the end of the track of which 16 will get used in this way. I think the drive refrains from writing over the 36 bytes of defect information (I'm not sure).
(Actually there is a switch which selects between bytes-per-sector and sectors-per-track, however the SMD adaptor documentation says that this switch must be off. Since nearly everything else in that documentation is incorrect, probably that is also.)

```
SMD Interface Lines

LG684 pin nomenclature, left-hand row of pins is 1-30, right-hand is 31-60.
Pins 1 and 60 are the same as in the more typical nomenclature, none of
the others are.

A-Cable (bus):                     (60 conductor)

1   31          Tag 1 -/+    (cylinder)
2   32          Tag 2 -/+    (head)
3   33          Tag 3 -/+    (control)
4   34          Bus 0 -/+
5   35          Bus 1 -/+
6   36          Bus 2 -/+
7   37          Bus 3 -/+
8   38          Bus 4 -/+
9   39          Bus 5 -/+
10  40          Bus 6 -/+
11  41          Bus 7 -/+
12  42          Bus 8 -/+
13  43          Bus 9 -/+
14  44          Open Cable Detect -/+    (also known as controller ready)
15  45          Fault -/+
16  46          Seek Error -/+
17  47          On Cylinder -/+
18  48          Index -/+
19  49          Ready -/+
20  50          Address Mark -/+    (not used, but is useful status on Fujitsu)
21  51          Busy -/+    (not used; we don't use dual-channel feature)
22  52          Unit Tag -/+
23  53          Unit 0 -/+
24  54          Unit 1 -/+
25  55          Sector -/+
26  56          Unit 2 -/+
27  57          Unit 3 -/+   (Tag 5 on Fujitsu)
28  58          Write Protected -/+
29              Sequence Pick L
    59          Sequence Hold L
30  60          Spare -/+    (Tag 4 on Fujitsu)
                        (This is where Bus 10 is most likely to be in the future)
                        (Priam uses this pair for that.)

B-Cable (radial):                  (26 conductor)

2   14          Servo Clock -/+
3   16          Read Data -/+
5   17          Read Clock -/+
6   19          Write Clock -/+
8   20          Write Data -/+
9   22          Selected +/-     (note inversion)
10  23          Seek End -/+     (not used on L Machine)
12  24          Index -/+      (not used on Priam drives & some others)
13  26          Sector -/+     (not used on Priam drives & some others)

1, 4, 7, 11, 15, 18, 21, 25      Ground


Some documentation says that Sequence Pick and Sequence Hold are unused on
the Priam.  It is lying.
```

```
Backplane connections for L-machine paddleboard (revised 11/20/81).
[Pin numbers to be assigned later, dependent on PC layout].
[Note that there is also a compatible Priam paddleboard.]
[This paddleboard also handles the console TV and perhaps the network.]

Outputs (to paddleboard):

Bus <9:0>
Tag <0:5>               Tag<4>=Bus<10>, Tag<5>=Unit<3>
Unit <2:0>
Write Data
-Write Clock
Read Clock Enb          = -Servo Clock Enb, controls Drive Clock
Power OK                controls Open Cable Detect, Sequence Pick & Hold

Inputs (from paddleboard):

Status <5:0>            (see below)
-Select Error           (prom looking at radial cable selected signals)
No Select               vs. Multiple Select
Index                   (from bus, not selected radial)
Sector                  (from bus, not selected radial)
Read Data
Drive Clock
Type <1:0>              0=SMD, 1=Priam, 2,3 reserved
-Index <3:0>            from radial cables
-Sector <3:0>           from radial cables

Total pins = 45


Status signals:
   0    Ready
   1    On Cylinder
   2    Seek Error
   3    Fault
   4    Write Protect
   5    Address Mark

Read Data is taken from whichever radial cable claims it is selected.
Drive Clock is Read Clock or Servo Clock from whichever radial cable claims
it is selected, according to Read Clock Enb.

There is no connection to Seek End.

Possibly there is an alternate paddleboard which provides for 2 drives
with separate bus cables to each.  The bus-cable receivers are open-collector
and enabled by the unit number.  This for drives like the M2312 which
lack bus-through capability (having only one connector).
```

```
Mapping from SMD paddleboard to Priam paddleboard:
[Note that the bus lines are actually tristated and bidirectional.]

Outputs (to paddleboard):

Bus <1:0>               NC
Bus <7:2>               Bus <7:2>
Bus <9:8>               -Ad <1:0>
Tag <2:0>               Head <2:0>
Tag <3>                 Rd
Tag <4>                 Wr
Unit <2:0>              Unit <2:0>
Tag <5>                 Unit <3>
NC                      Reset
NC                      Read Gate
NC                      Write Gate
NC                      Bus <1:0>       because of write gate, read gate
NC                      Bus OE L        enable paddleboard to drive Bus <7:0>
Write Data              Write Data
-Write Clock            -Write Clock
Read Clock Enb          NC
Power OK                NC              probably

Inputs (from paddleboard):

Status <5:0>            Bus <7:2>
NC                      Bus <1:0>       same pins as outputs for these 2
-Select Error           Ready
No Select               NC
Index                   Index
Sector                  Sector
Read Data               Read Data
Drive Clock             Drive Clock
Type <1:0>              0=SMD, 1=Priam, 2,3 reserved
-Index <3:0>            NC
-Sector <3:0>           NC

Total extra pins over SMD = 6
Total pins = 51
```

Network design notes

Table of contents:

```
Outline
  For wire-wrap version, build both Chaosnet & Ethernet interfaces
    For PC version, pick one, wedge both in, or cleverly share more hardware
    Actually, the Chaosnet-specific part is only the detector, transceiver
    interface, and myturn (if included).  Could be as few as 4 dips.
  Chaosnet
    Detect
      Sample at 20 MHz, see below.
      Carrier-sense counter
    Modulate
      8 MHz registered prom (actually 4 MHz, see ethernet)
      Does abort-generation also
    Myturn (could punt or could have microcode load register which
            is stuffed into counter at end of packet.  Punting probably OK?)
    Transceiver interface (2 dips + 2 resistors)
    CRC (use 9401 dogs?  PALs don't have enough internal state.  Doing in
         CPU takes 2 cycles per bit = 1.5 ms/max-size-packet.  Thus this
         is probably the best way to do it!  However if there is a CRC
         feedback shift register for ethernet, it can be jumpered to do
         the Chaosnet algorithm without too much trouble.  Note that
         the -error pattern is different, and Ethernet has complementation
         of first 32 bits of packet).
  10Mb Ethernet  (modulate, detect (PLL?), prefix, crc, collision abort)
    Detect (10 MHz phase-locked-loop & control)
    Modulate
      10 MHz state machine (20 MHz is too fast).  PROM output is 2 bits,
        external logic selects between them based on clock phase
      Jam signal can just be ordinary data apparently
      Preamble sent as ordinary data
    Transceiver interface
      Requires TTL/ECL conversion and at least one one-shot, sigh
    CRC (feedback shift register is about 9 dips, need 2 of them)
      Share CRC hardware by disallowing send-to-self, which is only
      useful for diagnostics?  Or by not checking CRC when receiving
      from self??  Do in CPU?  (Same speed as for Chaosnet, but less
      attractive since net is faster and we'll be using it more permanently).
    Retransmission delay in microcode
      Delay random multiple of 50 microseconds (up to 1000x)
      Requires ability to get periodic task wakeups (easy!)
    Frame deference delay hardware or microcode?  (9.6 us)
  Data in & out shift registers (32 bits each)
        (Amazingly enough, Chaosnet & Ethernet both send LSB first)
  Bus interface, task interface, FEP buffer interface
    Task wakeup conditions: input available, output ready, end of input
        packet, retransmit delay clock, output abort
  Output state: idle/busy/abort/send-last-bit.
  Input state: idle/read/await end-of-packet/synchronize PLL.
  Address recognition is in microcode and in FEP code
  FEP can monitor net without interfering with CPU use of net
```

Chaosnet detector:

Sampling at 5x bit rate, in the synchronized data stream the mid-cell edge occurs 2 or 3 clock intervals after the start-cell edge, and the start of the next cell occurs 4, 5, or 6 clock intervals after the start-cell edge. Thus the data is valid from 3T to 4T; since this is only one clock interval long, 5x is the slowest sampling rate that will work. At 4x, the mid-cell edge is at 1T, 2T, or 3T, and the start of the next cell is at 3T, 4T, or 5T, thus the two edges can get confused.

The detector's interface to the assembly shift register, the state machine which counts bits and makes DMA requests, and the receive CRC checker, is that it generates a CLK^ output which lasts from 1.5T to 3.5T, thus sampling the synchronized data between 3T and 4T when it is valid. Adding the 0.5 is necessary to provide guaranteed setup time, since clock and data are both coming from flip flop outputs clocked at the same time. (We could also use the opposite phase of the 20MHz clock somewhere to take care of this.)

The parts required (see my yellow notebook), besides a source of 20 MHz, are 5 D flip flops, 1/4 S86, 1/3 S10, 1/4 S02, and something to make a delay of 20 ns or so. Normal registered PALs are too slow, but an ALS16L6 could do this, otherwise it takes roughly 3 DIPs (portions of 5). The S86 and S10 can be replaced with an S64, which doesn't buy much.

Ethernet detector:

This pretty much has to be a phase-locked-loop. I don't think the same PLL can be used for Chaosnet, as Chaosnet gives it no time to lock onto the phase of the transmitter, and even at the slow Chaosnet read it must take several clock transitions. [IS THIS REALLY TRUE?]

The same assembly register, receive state machine, and CRC register are used as for Chaosnet. The clock however comes from the PLL instead of from the detector.

```
Chip-count estimate:

Chaosnet detector:               3  (S175, 1/2 S74, various S gates)
Chaosnet carrier sense:          1  (LS161)
Modulate:                        2  (256x8 registered prom?, ext logic)
(Myturn not done)
Chaos transceiver interface:     5  (drive, receive, term, loopback, misc gates)
Ethernet detector:               ?  (assume 8 I guess, some ECL?)
Ethernet transceiver interface:  ?  (assume 5 I guess, some ECL.)
2 CRC feedback registers:        18 (LS374, LS86, some kind of comparator)
                                 2  (for underestimate of CRC control logic)
Ethernet carrier sense, defer:   2  (some sort of one-shot or something)
(retransmission delay not done, probably part of memory refresh or something)
Data in & out shift regs:        8  (LS374)
Data in & out buffer regs:       8  (LS374)
FEP datapath buffers:            4  (2 LS374, 2 LS244)
Task & FEP handshake:            ?  (assume 4 I guess)
Overall state and control:       ?  (assume 5 I guess)
(bus interface not done)

TOTAL                            53 + 22 assumed

This is too big; cut down from 75 to 50 somehow.  Making it half duplex
would save about 20 at the cost of making 1-machine testing impossible.
Also the assumed figures may be too high.  Another possibility is to
make it impossible to transmit while using the disk.

Using 74LS595, 74LS597 for shift/buffer registers saves 8.
```

Loose ends:

Task & FEP handshake

Phase locked loop (I don't know much about these)

Find right part for transceiver interface (Ethernet spec is obvious BS)

Design "overall state and control", which involves commands coming in from bus, the input and output states mentioned above, and certain delays needed by the Ethernet.

Use bidirectional transceivers with registers for the bus interface?

Tasks

There are three tasks involved. The output task is high priority and does DMA for outgoing packets. The input task is high priority and does DMA for incoming packets. If the interface is half-duplex, the input and output tasks are the same. The input task does not look at the incoming data (at 10 Mb, it comes in too fast if other things are happening at the same time, e.g. a disk transfer.)

The service task is low priority and shared with other DMA devices. When input or output completes, the service task is awakened by the appropriate DMA task. The input task also wakes up the service task after reading the hardware header words of a packet; the input task continues reading the rest of the packet while the service task runs in the background. The service task finds the next input or output packet buffer and sets up the address and word count. When the header of an incoming packet has been read, the service task looks at the destination address and decides whether it is for this machine (by direct transmission or by broadcast/multicast). If the packet is not for this machine, the input task's state is smashed and the hardware is directed not to send any wakeups until the start of the next packet. The DMA address and word count are set back to reuse the same packet buffer. If the packet is short and has already been read before the service task decides it is not to be received, it is possible to miss the next packet, but the buffer lists can't get screwed up since only the service task manipulates them.
This is well worth doing (rather than receiving all packets in their entirety), because the DMA task for an incoming packet uses about 1/8 of the machine.

There needs to be some sort of queue of requests to the service task in A memory. It deals with buffer lists and command lists for each DMA device, which are stored in main memory (using physical addresses, not virtual addresses). Lisp code manipulates these buffer lists, and the service task and the DMA tasks it controls acts like an autonomous I/O channel to the Lisp code.

Second-draft design for L disk control. -*-Text-*-

Table of contents:

Functional spec

Functions are divided into three categories according to their real-time constraints:

Unit selection, seeking, and miscellaneous things like recalibration and error-handling are done by Lisp code. There are I/O device addresses (pseudo-memory) which allow sending commands to the disk drive and reading back its status (and its protocol, e.g. SMD, Priam). When formatting the disk, the index and sector pulses are directly read from the disk through this path (they are wide enough!), and the timing relative to them is controlled by Lisp code or special formatting microcode.

Head selection is the same except that it is done by microcode rather than Lisp code so that an I/O operation may be continued from one track to the next in a cylinder without missing a revolution because of the delay in scheduling a real-time process to run some Lisp code.

Read/write operations are done by disk control hardware in cooperation with microcode. There is a state machine which generates the "control tag" signals to the drive (i.e. read gate and write gate), controls the requests to the microcode task to transfer data words into or out of main memory, and controls the ECC hardware.

When the FEP is using the disk, the first two functions above are performed by LIL code in the FEP; the third function is performed by the disk state machine in cooperation with the FEP's high-speed I/O buffer.

The disk state machine can select its clock from one of two unsynchronized clocks, both of which come from the disk. One is the servo clock and the other is the read clock, derived from the recorded data. Servo clock is always valid while there is a selected drive, it is spinning, and it is ready. Delays are always generated from the servo clock, not from the machine clock or one-shots. [Contrary to a previous version of this file, the state machine is never clocked by the Lisp machine clock.]

The state machine is started by an order from the microcode, Lisp code, or the FEP and usually runs until told to stop (thus there are no sector-length counters in the hardware). When an SMD is being used, most of the lines on the disk bus, including control tag, come from a register which must be set up beforehand, but the Read Gate and Write Gate lines are OR'ed in by the state machine.

The state machine stops and sets an error flag if any of the following conditions occurs:

    No disk selected (SMD)
    Multiple disks selected (SMD)
    Disk not ready (Priam)
    Overrun (slow response from microcode)
    An unexpected index or sector pulse
    Writing the command register while the state machine is running

These error checks prevent clobbering an entire track if the microcode dies for some reason and never sends the stop signal.

Other errors from the disk, such as Off Cylinder, are not checked for. Most drives will cause a fault if any error occurs while writing. The disk error status (including fault) is checked by microcode and by macrocode after the sector transfer is completed.

The state machine can hang if the clocks from the disk turn off for some reason. The macrocode should provide a timeout.

The following orders to the state machine exist, i.e. it has the following programs in its memory:

[These are somewhat obsolete in the fine details and need to be reexamined.]

Read. The state machine delays, turns on read gate, delays some more, changes from the internal clock to the disk bit clock, waits for a sync pattern, then reads data words and gives them to the microcode until told to stop. The stop signal is issued simultaneous with the acceptance of the third-to-last data word by the microcode task. After reading the last data word, the ECC is read, and the microcode task is awakened one last time as the state machine goes idle. The microcode reads the ECC=0 flag over the bus; the flag is 1 if no error occurred.

Read Header. The state machine waits for a sector pulse, delays, turns on read gate, delays some more, changes from the internal clock to the disk bit clock, waits for a sync pattern, reads one data word (a sector header), turns off read gate, and falls into the Read program. The header word is given to the microcode as data (32 bits of header and 4 bits of garbage); it is up to the microcode to do header-comparison to make sure that the proper sector is being accessed. There is no ECC on the header, instead there are some redundant bits which the microcode checks in parallel with the real bits. In other words, the header consists of 6 bits of sector number, 6 bits of head number, 12 bits of cylinder number, and 4 bits of some hash function of the other bits, fitting into the 28-bit header stored in a DCW list (see below).

[At the moment (7/15/82) Read Header has been flushed and there is just Read. This may need to be changed...]

Write. This writes the preamble for (the data portion of) a sector, then the data, then an ECC code of the data, then a guard byte. Before giving this order give 0 as a word of data (which will become the first word in the preamble). The state machine turns on write gate without delaying, writes the given word of data, requests another, and continues writing until told to stop. It then enables the ECC register and continues requesting and writing words until told to stop again, at which point it writes the ECC code and a guard byte and then stops. The first stop signal is given along with the first data word (preceding words are part of the preamble, all 0 except for the last which has the high-order bit 1). The second "stop" signal is given in lieue of a data word when one is requested. This order is used for writing the data portion of a sector and for formatting the disk. When writing a sector, the Read Header order is used and then after the header word has been checked the state machine is forcibly stopped (it is now doing a Read order) and given a Write order.

[Actually this order requires only actual data to be given by the microcode; the state machine supplies the necessary number of 0 bits as preamble, followed by a sync pattern (four 1 bits).]

Read FEP. This is just like Read Header (and Read) except that it is used by the FEP, which affects the details of DMA handshaking and the word length (36 vs 8). The FEP does not write (in this scheme it could not order the state machine into Write in real time after completing Read Header). The FEP receives the header followed by the data; it must do the header comparison and read again if the wrong sector was read. After the transfer the FEP reads the ECC=0 flag over the bus to see if an error occurred.
[H - Could it be arranged so that the front end can read the previous sector's header and enables write for the following target sector? I can see that patching up the disk, or some other diagnostic write functions might come in handy. This special write command would be locked out if not given during the Read order before the next sector pulse.]

[Moon - This is a good idea. Should be a new order which reads a header, then delays until sector pulse, then reads and ignores a header, then writes. During that long delay the FEP buffer gets turned around. There needs to be some positive signal that the FEP verified the header and didn't just die, with no race conditions that can lead to writing the wrong sector (although you could start by reading the whole track so that if you clobbered something you could put it back!). I don't think there's any way to coerce the existing hardware into doing this, but one new FEP-settable bit, which the state machine could test (die if not set) would do the job.]

[Moon - Actually, I'm not sure how important this is. If the FEP can load the microcode, and the machine works, it can use the microcode to write on the disk out of a buffer in main memory. This is probably good enough for patching up the disk. The only reason the FEP has to be able to read from the disk on its own is so that it can load diagnostics when the processor is broken or when it doesn't want to disturb the state of the processor.]

[Moon 3/82 -- the ability for the FEP to write has been flushed, at least for the moment. It could be put back, but doesn't seem to be worth the extra parts.]

Error Correct. The servo clock from the disk is used. The state machine cycles the ECC feedback shift register, generating bogus data requests to the microcode, and the microcode counts words. First the state machine sends a short word, then it sends long words until the microcode says to stop; this runs the right number of times to make the ECC code repeat. Now the state machine continues sending words, which the microcode counts, but enables its clock to stop when the ECC register contains a zero pattern which indicates that the error has been found. At this point the microcode reads out the error burst bits and the bit position within the word. In the case of an uncorrectable ECC error, the microcode will give up after a certain number of words and reset the state machine. When a Read terminates with an ECC error, the microcode issues an Error Correct order and deals accordingly with the results. [The FEP does not support error correction, only error detection? Or maybe it can use Error Correct and see how many bytes come into its buffer?]

There are also slightly modified orders Read32, ReadHeader32, and Write32. These are just like the corresponding ones except that the data consist of 32-bit words, right-adjusted within the machine's 36-bit words. When reading, the first word is given 36 bit clocks rather than 32, so that the first word is right-adjusted in the shift register. The last 4 bit clocks of the last word read overlap with checking of the ECC code; this is no problem. When writing, there is no problem, since everything is naturally right-adjusted anyway. I'm not sure whether the disk control will contain hardware to plug in fixnum data type in the high 4 bits (rather than the low 4 bits of the next word) or whether the processor can just go back and fix up the data types later. The 32-bit I/O mode is used to allow the file system to avoid wasting 11% of the disk. (On Fujitsu disks, there are 16 sectors per track in 36-bit mode, and 18 sectors per track in 32-bit mode.) It is also used to make it possible to transfer 36-bit virtual memory images over the network as streams of 32-bit words.

Normal read is a Read Header order which automatically turns into a Read order after reading the header. The microcode compares the header against the desired header; if they differ, it aborts the Read and issues another Read Header which will read the next sector's header. It is desirable to have accurate timing of the turning off and on of read gate, so this is done by the disk state machine rather than the microcode.

Normal write is a Read Header order which is aborted and turned into a Write order after reading and checking the header. The microcode supplies the appropriate length sync pattern before supplying the data words. The decision to write cannot be made in the disk state machine, as we must not write if the header fails to equal the desired sector. Thus the microcode must decide whether to do Write or another Read Header.

Write All is done by manually searching for an index pulse then issuing a Write order. The extraneous ECC code written at the end of the track doesn't matter. (On a Priam, we first wait for an index pulse then wait for a sector pulse before writing, in order to skip the bad-track information).

Read All is done by manually searching for an index or sector pulse then issuing a Read order.

[Actually there need to be separate orders for Read All and Write All just to inhibit the error for sector pulse in the middle of an operation.]

[I haven't bothered with Read All since it doesn't seem to work on the A machine anyway. You can't read reliably across a write splice, because the VFO loses sync and doesn't regain it, at least on Tridents.]

[9/82: Well, I put Read All back in anyway. It probably only works to read from the index pulse up to the first following write splice. This is enough to be able to read media defect records, also possibly to do a little checking of formatting.]

Sector Data Format

Not all of this is wired into the hardware.

A sector contains 9280 data bits. In 32-bit mode, this is transferred as 290 32-bit words, each with fixnum data type. By convention, the first two 32-bit words are used as a tag, allowing reconstruction of the front-end file system by reading the whole disk. Formatting the disk writes 0 into the tag words (actually into all the data words).

In 36-bit mode, a sector is transferred as two 32-bit words (the tag), followed by 256 36-bit words, typically containing the data for a page. Unfortunately, when reading in 36-bit mode the hardware cannot put fixnum data type on the tag words; they get "whatever bits are there". By convention the tag words are written with 0 and "PAGE" (4 ascii characters) when a page is written in 36-bit mode.

Macrocode to Microcode Interface

[Brought up to date with the microcode and Sysdef, 9/9/82]
[Updated for IO-REV2A ECO#4 5/7/83]

The read/write microcode is driven by a DCW (disk control word) list stored in main memory. The Lisp code which builds the DCW list and interprets the results, as well as doing the seeking and unit selection, is wired down in main memory and called by the page-fault handler, which is written in Lisp. The boot sequence which initially gets this stuff into main memory involves a DCW list built by the FEP, which reads the first nK of the selected band into the first nK of main memory. A DCW list can specify any number of transfers within a single cylinder. Transfers which cross cylinder boundaries are done as multiple separate transfers as far as the microcode is concerned. Skipping of bad sectors and bad tracks is done at the Lisp code level.

The DCW list consists of a sequence of command blocks in consecutive physical addresses. Any number of command blocks can be specified, however they can only operate on a single cylinder of a single unit since the slow operations of unit selection and seeking are performed by Lisp code. A DCW can request that Lisp code be awakened (via a sequence break), and the address of the currently-executing command block is in a Lisp-accessible A-mem variable, %DISK-DCW-ADDRESS.

The information in the DCW list is heavily dependent on the needs of the microcode, rather than being a general device-independent channel interface. Such an interface is provided by Lisp code, not by the microcode.

The microcode is divided into two tasks: a high-priority DMA task (%DISK-DMA-TASK), which transfers words between disk and memory and also handles header processing at the start of a sector; and a low-priority service task (%DEVICE-SERVICE-TASK), which interprets DCW's and passes parameters to the DMA task.

I expect that other DMA devices will use a similar interface and will share the same service task.

Any disk error, including a soft ECC error or a compare failure in read-compare stops execution of DCW's and wakes up the disk driver, which can read the address of the current command block, the last memory address referenced by data transfer, a software status word, and the hardware status. All retries are handled by Lisp code, not microcode. When the microcode stops, the hardware status is left undisturbed.

The first word in a disk command block is the disk control word (DCW) itself. It is a fixnum containing the following fields:

```
        %%dcw-micro-command     4 bits  Microcode command opcode, %dcw-u-xxx
        %%dcw-dcr-command       4 bits  Value to go into %%dcr-command (see below)
        %%dcw-length            6 bits  Total number of words in command block
The following additional fields are not used by the microcode.
        %%dcw-command           6 bits  Command opcode, %dcw-xxx
        %%dcw-n-addresses       6 bits  Number of address/word-count pairs (DAPs)
        %%dcw-class             4 bits  Who this command is for
                                2 bits  (spare)
```

Some DCW command opcodes are executed entirely by Lisp code in the disk driver, and never seen by the microcode.

```
        %dcw-unit               2nd word is unit number of drive to be selected
        %dcw-cylinder           2nd word is cylinder number to seek to
```

The DCW command opcodes trivially executable by microcode are as follows. %dcw-xxx is implemented by %dcw-u-xxx.

```
        %dcw-nop                does nothing
        %dcw-stop               end of DCW list, implies wakeup
        %dcw-wakeup             awaken the disk driver and continue execution
        %dcw-seek-wait          wait for disk seek to complete, then continue execution
                                2nd word is what goes in command register to wait
        %dcw-goto               2nd word is physical address of next DCW
        %dcw-head               2nd word is disk command to select head (the microcode
                                will turn the head-tag bit on and off)
```

The remainder are data-transfer commands (see below):

```
        %dcw-read               Read 36-bit words
        %dcw-read32             Read 32-bit words
```

%dcw-write	Write 36-bit words
%dcw-write32	Write 32-bit words
%dcw-read-compare	Read 36-bit words, compare to memory
%dcw-read-compare32	Read 32-bit words, compare to memory
%dcw-write-all	**Write 32-bit words through entire track (for formatting)**
%dcw-read-all	Read 32-bit words at start of track (mainly for reading media defect records before formatting)
%dcw-read-header	Read just the header of one sector

%dcw-xxx and %dcw-xxx32 are both implemented by %dcw-u-xxx.

There is also %dcw-u-ecc, see below.

Non-microcode-executable commands have %dcw-u-stop in the %%dcw-micro-command field in case the microcode ever sees them.

In a data transfer command, the 2nd word is a fixnum containing the expected sector header. Each data transfer command transfers a single sector. A sector is 2 32-bit tag words followed by 256 36-bit words or 288 32-bit words in the standard format, however the microcode does not itself require any particular sector size.

The 3rd word is the value to be stored into the disk-command-register to start the transfer; this includes the bits to go on the disk bus, the proper disk state machine program number (%DCR-COM-xxx), the start bit, and the proper task number. For reads, there is only one disk state machine program to be executed and its number is in the %%dcr-command field of this 3rd word. For writes, read-all, and read compare, two state machine programs are used: one to search for the proper sector and a second to transfer (write or compare). The %%dcr-command field of the 3rd word contains the search program (Read), while %%dcw-dcr-command contains the transfer program. For reads, %%dcw-dcr-command should contain the same value as the command in the third word.

The remainder of the command block is a sequence of pairs of words. Each pair is called a DAP (disk address pair). The first contains a number of words to transfer from sequential addresses and the second contains a starting physical address. The sign bit of the second word is set if another DAP follows (this is called the "chain bit"). There are some funny things about the word count: In most DAPs, the word count is 2 less than the real word count; this implies that each DAP must transfer at least 2 words. In the last DAP for a read or read-all (but not read-compare), the word count is 4 less than the real word count; thus the last 4 words must be read into consecutive physical locations. In the last DAP for a write or read-compare, the word count is 3 less than the real word count.

These restrictions will be handled by requiring that disk buffers that might lie across page boundaries (and hence be in non-contiguous physical memory) must be aligned to quad-word boundaries, so no DAP need transfer fewer than 4 words. Note that in 32-bit mode a disk buffer is bigger than a page. Alternatively the disk driver could copy a few words between the user's buffer and an internal buffer, when necessary due to a page boundary falling near the beginning or end of the buffer, but this seems unnecessary.

The fields in a sector header are:

%%disk-header-sector	6 bits	Sector number within track
%%disk-header-head	6 bits	Head number
%%disk-header-cylinder	12 bits	Cylinder number
%%disk-header-pack	6 bits	Hash code of physical media serial number
%%disk-header-spare	1 bit	must be zero for now
%%disk-header-parity	1 bit	Makes the 32-bit word have odd parity

The cylinder number verifies that the disk seek worked. The head number verifies that the correct head was selected. The sector number distinguishes the several sectors on a track (rotational position sensing is optional and not relied-upon). The pack number verifies that the correct unit was selected and that the correct pack is mounted on it (for demountable drives). The parity bit prevents a dropped bit in the sector field from causing the wrong sector's header to match.

%dcw-read-header is a special case of a data transfer command. The sector header (second word in the DCW block) is ignored. The DAP count and chaining are also ignored. A single word is stored at the address pointed to by the DAP and then the command terminates; this word is the header of the first sector encountered. The disk command register value (third word in the DCW block) should be a Read command.

NOTE: The above is true as of microcode 210. Previous microcode versions worked differently, and strangely.

The microcode status readable by Lisp code is contained in the following special variables (mapped into a communication area in A-memory):

```
        %disk-dcw-address       physical address of the first word in the DCW
                                command block currently being executed.

        %disk-micro-status      status of the microcode tasks, intercommunication
                                between the two tasks, and communication with
                                the Lisp-coded driver.
            %disk-micro-status-idle             Nothing happening
            %disk-micro-status-start            Starting up at address in %disk-dcw-address
            %disk-micro-status-in-sector        Transferring data
            %disk-micro-status-end-read         Sector read, must check ECC
            %disk-micro-status-end-write        Sector written
            %disk-micro-status-end-read-compare Sector read-compared, must check ECC, r/c flag
            %disk-micro-status-end-sector-wait  Sector-pulse found
            %disk-micro-status-search-error     Error: matching sector header not found
            %disk-micro-status-disk-error       Error: some disk hardware error
            %disk-micro-status-stop             Normal halt: stop DCW executed
            %disk-micro-status-ecc-done         Ecc computation completed

        %disk-wakeup            normally nil, set to t when disk driver to be waked
                                by wakeup DCW, stop DCW, or error

        %disk-memory-address    Current physical address for the DMA task.
                                When DCW execution terminates, this is the last
                                address referenced+1.

        %disk-command-address   Physical address of the disk command register in the IOB.
                                This is a locative pointer in virtual=physical space.

        %disk-status-address    Physical address of the disk status register in the IOB.
                                This is a locative pointer in virtual=physical space.

        %disk-sector-max-tries  Maximum number of sectors to check before giving a
                                search error. Must be more than the number of sectors
                                per track. Set this to "infinity" for debugging.
```

The microcode function %disk-start wakes up the service task, which will start executing DCW's at the address in %disk-dcw-address. It is necessary to set %disk-micro-status to %disk-micro-status-start first.

When an ECC error occurs, DCW execution is terminated. The disk driver starts a new DCW to get the disk control (hardware and firmware) to compute the error pattern and position. This DCW has %DCW-U-ECC; it's second word is a %DCR-COM-ECC command, with the busy bit on, no disk tags except unit tag, and the %DISK-DMA-TASK. This DCW need not be followed by a stop DCW; it will terminate with %DISK-MICRO-STATUS-ECC-DONE; %DISK-MEMORY-ADDRESS will contain a fixnum which is the number of iterations executed by the microcode. If this is strictly greater than 145., it is a hard ECC error.
Otherwise, the ECC pattern and position register in the hardware should be consulted. The position register contains the low 7 bits of the bit position in the sector where the least-significant bit of the error pattern appears. The pattern register contains 11 bits starting at that bit position. To compute the high bits of the bit position, use %DISK-MEMORY-ADDRESS which is in multiples of 64 bits and is either correct or too large by 1; decrement it if necessary to make it agree with the high bit of the ECC position register.
The contents of the pattern register should be xor'ed into the data starting at that bit position. The driver has to deal with splitting the pattern across word boundaries, and with the 32-bit vs 36-bit word distinction. After recovering the data, the driver should resume execution in the original DCW list with the next DCW block after the one that got the ECC error.

Note that a hard ECC error will cause an overrun. The overrun can always be ignored if the biword number is > 145; if the biword number is <= 145, and overrun status is true, it was a real overrun and the ECC computation has been spoiled.

In 36-bit mode, care must be taken when converting the bit number in the sector into a word-address/bit-number in memory. The two tag words at the front of the sector contain only 32 bits each. Care must also be taken for an ECC correction in the last word of a sector, where the pattern might lap over into nonexistent data (those bits of the pattern should be zero).

Notes about crocks.

Rotational position sensing, originally intended to be flushed, has been put back in since it only takes 4 DIPs on the main board and 4 on the paddle board. Note that Fujitsu drives do RPS in the drive; there is a sector counter that can be read back using the extra tag lines. If the paddle board is tight for space we could flush RPS again. Some SMD's (e.g. CDC ones) do not drive the "B" cable index and sector lines, so RPS will not work for them. RPS will not work for Priams and probably not for Tridents (if we make paddle boards for those drive interfaces.)

We will support the extended status readback of the Fujitsu drives, which lets you see the drive type and options, all the unsafe conditions, etc. Too bad the Eagle is not compatible with earlier Fujitsu drives.... This should be quite useful for maintenance.

Overlapped seeks are supported but there are no attentions (the SMD feature for this is brain-damaged, and the Priam doesn't have it at all.) Instead when the disk controller is idle, but seeks are in progress, the disk task can wake up on each sector pulse from the selected drive (i.e. about once per millisecond) and poll the seeking drives, looking for ON CYLINDER or for SEEK ERROR. The overhead on the machine to do this is determined by the speed at which you can select drives as specified by the SMD specs; probably it will be about two microseconds per drive polled.

The machine will support both the SMD interface and the Priam interface, using identical disk control boards, but different paddle-boards (electrical and physical interface) and different microcode and macrocode. Probably this will not be a "sysgen option", but will be figured out at boot time (desirable to transport software images en masse from machine with one kind of disk to machine with the other kind). The paddleboard is the board which sticks onto the pin side of the back-plane and contains line drivers and attaches to the cables to the disk (which pass through another set of connectors at the bulkhead).

We might want to think about whether we can support the Trident interface also. This is discussed below.

```
Revised attempt at module breakdown  [semi-obsolete]

Bus Interface
        Task request
        DMA data in & out
        Overrun detection
        SPY DMA data out (& in?)
        Lbus Dev address decode
        Memory-mapped I/O decode?
        Spy bus decode, to start read????
            Card ID match Lbus Addr, spy write to special reg selects DMA dev
        Reset
        (stuff shared with TV and Network)

Disk Interface
        Edge connections
        Paddleboard drivers (register) & receivers
        Rotational position counters
        Channel Ready, Sequence (from Lbus Power Reset I guess)

Serial data path
        Shift register
        ECC register (and any counter?)
        Preamble recognize, clock-switching logic
        Read/compare capability?

State machine
        Prom, registers, PAL decode?
        Bits per state (word) counter
        Wait conditions (index/sector, preamble, word, disk ready?)
        Error terminate?
```

Microprogramming interface

"Memory-mapped" I/O is used for all functions except those relating to the DMA task. This allows the FEP to read from the disk simply by doing Lbus operations, with no need to execute microinstructions (the CPU however must be stopped or at least known not to be touching the disk itself). No provision is made for the FEP to use the disk when the Lbus is non-functional. (The FEP can use the network without using the Lbus at all, for purposes of remote maintenance within a site as well as remote maintenance using a single dialin line shared between all machines at a site.)

Command Register

This register directly controls the bus, tag, and unit-select lines to the disk(s), provides a DMA task assignment, and selects a state-machine program to be executed. If the state machine is running when the command register is written, it is stopped with an error. Otherwise it may optionally be started (if bit 24 is 1). Writing the command register resets various error conditions. All bits in the command register may be read back. All bits in the command register except the low 8 are zeroed by Lbus Reset. [Note that this register is distributed over the DKPDL and DKTASK prints.]

10:0	Disk bus. (On Priam, there is only an 8-bit bus and the high 3 bits are the head number.) On Fujitsu disks, bit 10 is tag-4. On Priam, bits 7:0 may read back differently than they were written if the drive is sourcing the 8-bit bus.
11	Dbus in. On Priam, enables the disk bus to be driven by the disk rather than the controller.
15:12	SMD: tags 3:0. Priam: RD, WR, -AD1, -AD0.
19:16	Unit number. On Fujitsu disks, bit 19 is tag 5.
23:20	Command opcode (selects state machine program).
24	Start. Starts state machine if 1. Reads back as -DISK IDLE (1 if state machine running).
28:25	Task. 8-15 selects that task, otherwise no task.
29	FEP using disk. Enables SPY bus DMA.
30	36-bit mode (off forces fixnum data type in high bits)
31	Fast disk mode (set to 1 for M2351)

To select a unit on SMD, put the unit number in bits 19-16 and 0 in bit 12 (tag 0). Then change tag 0 to 1. On some disks the unit number no longer matters after this, on others it must not be changed. To select a unit on Priam, just put the number in bits 19-16.

To initiate a seek or a head select on SMD, put the appropriate number in the bus, then set tag 1 or 2 to 1, wait a microsecond, then clear the tag. To perform special operations (recalibrate, fault clear) on SMD, do the same thing with tag 3.

To perform non-read/write operations on Priam, (e.g. seek, recalibrate, sense drive type, sequence up or down, etc.), use bits 15:11 of the command register to send commands to the Priam, and bits 7:0 as the data (when writing the data bus).

During read/write operations on an SMD, the disk bus should contain any modifiers desired (e.g. servo offset) and tag 3 (control tag) must be turned on.

A task wakeup occurs if the state machine orders one, and whenever the state machine is not running. No task should be assigned by the command register when the state machine is not being used. A wakeup will always occur immediately when a task assignment is given.

The fast disk mode is simply a bit that the state machine microprogram can branch on. It should be set when selecting a unit to the appropriate value for that unit, which can be read from its label at the same time its geometry (number of heads, etc.) is read; that read should try both values of the bit although for reads from the FEP this probably makes little or no difference. Fast disk mode adjusts the amount by which a task wakeup preceeds the need for data, and adjusts the vfo sync delay and the write splice delay. What it really does is to select between two sets of state machine microprograms, allowing drives of two incompatible types to be intermixed.

Diagnostic Register

This register allows a program to disable the paddle board and simulate a disk, testing most of the logic with the machine fully assembled. This register is cleared when the machine is powered on.

0	Read clock
1	Servo clock
2	Read data
3	Index
4	Sector
7:5	(spare)

Paddle Enable Register

This register is cleared when the machine is powered on. It allows the paddle board to be turned off. It is set to 10 for normal operation. The bits are:

0	Paddle ID enable (paddleboard ID prom to disk bus)
1	-Paddle disk enable (disconnect disk part of paddle board)
2	-Paddle net enable (disconnect network part of paddle board)
3	Paddle power OK (enable disk to spin up)

Status Register

Reading this register reads the status of the selected drive, of the disk interface, and some internal diagnostic signals.

5:0	SMD: disk status (see below). Priam: not used
6	Index (a narrow pulse)
7	Sector (a narrow pulse)
8	Paddle ID enable
9	-Paddle disk enable
10	-Paddle net enable
11	Paddle power OK
12	Error (any error that stops the state machine)
13	SMD: Select error, Priam: Selected drive not ready (or no drive)
14	Overrun
15	ECC=ZERO (1 after a read operation if no error)
16	Read compare (1 if disk and memory don't agree)
17	End flag (diagnostic)
18	Buffer busy (diagnostic)
19	Wakeup (diagnostic mostly)
23:20	(spare)
24	Write data (diagnostic)
25	Next state 0 (diagnostic)
26	Advance state (diagnostic)
31:27	State. Address of next state machine microinstruction to be executed. (diagnostic)

Overrun and Error are cleared by writing the command register (however writing the command register while the state machine is running will set Error and stop the state machine).

The SMD status bits are as follows: (these can be modified by tags 4 and 5 on Fujitsu drives)

0	Ready
1	On Cylinder
2	Seek Error
3	Device Check (Fault)
4	Read Only
5	Address mark found (not used)

Rotational Position Sensing

This is a 16-bit register with 4 bits for each drive, containing the current sector number. This does not work for Priams and some SMD's. This register is not synchronized; it should be read in a loop until two successive values agree. For formats with more than 16 sectors per track, the counter will stick at 15 during the extra sectors.

Error Correction

If bit 15 of the status register is 0 after a read operation, an ECC error was detected. The error-correct state machine operation may be used to compute the error syndrome. The microcode task wakes up every 64 bits, simply to count the bits. After the state machine stops, the error correction register may be read:

10:0	Error pattern
17:11	Bit number within the 128-bit word

How to use these data was described above.

It may work for the FEP to use the error correction state machine program, maybe.

DMA transfers

A microdevice write operation is done during the address cycle. At the same time the sequencer is told to dismiss the task and the memory control is told to start the appropriate (read or write) DMA cycle.
Bits in the Lbus device address are:

9:5	card slot number
4:3	subdevice (0 = disk)
2:0	operation

Operations:

0	write disk buffer directly (rev 2 and later)
1	dma cycle (start dma cycle without dismissing)
2	dismiss, task acknowledge (just clear wakeup)
3	dismiss & dma cycle
4	dismiss (only)
5	kill disk task
6	dismiss, task acknowledge, set end flag
7	dma cycle & set end flag & dismiss

Operation 3 is what is normally used. Operation 1 could allow transferring multiple words per task wakeup if there was more than 1 word of buffering; it is also probably needed by the microcode in order to start a DMA transfer for the disk while continuing to run the task.
Operation 2 is used for non-data-transfer task wakeups, such as the wakeup on sector pulse and the wakeups used to count words when doing ECC correction. It simply dismisses the task (clears wakeup), and also has different timing with respect to the Overrun error.

Operation 5 clears the disk task assignment, preventing further wakeups, clears control tag so that the next disk command can be given cleanly, and also "accidentally" clears fep-using-disk and disk-36-bit-mode.

When reading from disk into memory, after the dma cycle with the end flag there will be two additional data words; the state machine will then read and check the ECC code and then stop.

When writing from memory to disk, the data word supplied with the end flag is the second-to-last data word in the sector; the state machine will accept one more data word, then write the ECC code after it, write a guard byte, and then stop. The same timing applies for read-compare.

For microdevice read, the bits in the Lbus device address are:

9:5	card slot number
4:3	subdevice (0 = disk)
2:0	operation (0 for disk = read data buffer)

A microdevice read from the disk reads the data buffer. This is used to read the header of a sector (otherwise it would have to be written into memory with a DMA cycle and then read back.)

DC-based breakdown [This page is obsolete]

More detailed breakdown of the disk control, based on looking at the CADR DC:
Very rough chip-count guesses in <> brackets.

Error flags (DCBUSY) <6>

Interface for non-real-time functions (DCREG, DCSTS, DCXBSA, DCXBUS) <20>
Includes bus drivers and receivers (share with other devices?)

Microcode task wakeup condition (small parts of DCCHAN) <3>
FEP I/O Buffer handshaking <5>

State machine clock selection (small part of DCCLK) <5>
State machine order interface (part of DCCMD) <2>
State machine data interface (parts of DCRBUF, DCSH, DCWBUF) <12>
State machine itself (parts of DCUC, DCUI) <15>

ECC Feedback Shift Register (DCECC, part of DCPOSC) <12>
(Unless this is done with a chip such as AMZ8065).

Drive interface (DCDBUS) <5>
(Note that the drivers and receivers, and almost all of the difference
between Priam and SMD, is off-board on a bulkhead adaptor board. For
the differences between Priam and SMD we will use a different state
machine program, different Lisp and FEP code, possibly slightly different
microcode, and unfortunately several jumpers or other personalization
of the logic.)

Irrelevant prints (DCCAPS, DCCCW, DCCLP, DCDA, DCEDGE, DCHDCM, DCPAR,
most of DCPOSC, DCTMOT, DCTRID, DCTRSG, XBUS)

Very rough total parts count = 85
Hopefully this is an overestimate and things can be designed more
cleverly than in the CADR, as well as deleting some portions of
the CADR design. 60 is certainly a plausible number.

```
New revised parts estimate, based on actual logic sketches:

State machine: 13
        PC and clock counters, prom, pal, clock selection.
Shift and buffer registers: 18
        Note that the shift register is a 40-bit register.  Processor
        inputs to the low 36 and outputs from the high 36.  FEP inputs
        to the low 8 and outputs from the high 8 (16?).
   [Using 74LS595, 74LS597 saves 5 DIPs (also 16-pin instead of 20)]
   [These parts aren't available until November, however.]
   [Another possibility is a 40-bit 74LS299 wired to a 40-bit
    bidirectional registered transceiver, which acts as buffer register
    and maybe bus transceiver.  Extend the 32-bit-mode hack to allow
    storing 36-bit words in this 40-bit register.]

Disk interface (both state-machine and non-real-time): 10 (?)

ECC Feedback Shift Register and data interface: 10
Task/FEP handshake:  5?

New total: 56

Bus interface (shared with other devices on board): ?
  Data interface, address receive, address decode, control handshake

As of 1/3/82 it is 65 dips, not counting the shared portion of the
bus interface, most of which is shared with both NET and TV.
```

Paddle boards:

There are two paddle boards, SMD and Priam. The backplane interconnection to the disk control board is arranged to be compatible between these two paddle boards, which requires 11 extra pins in the Priam case and 6 in the SMD case. The paddle board plugs directly into the backplane pins opposite the board with the disk control on it, and provides flat cables to the bulkhead. At the bulkhead a connector provides strain relief and shield grounding for the external cables to the disk.

[We won't be doing a Priam paddle board initially. There is also the possibility of a Trident paddle board.]

SMD paddle board:

The SMD paddle board provides 4 to 6 radial cables, however many turn out to fit.

The following variations from standard SMD are provided for:

The B-cable signal Seek End is not used. Seek End is not useful enough to be worth the trouble required to turn it into Attention. Also the Index and Sector signals, which only exist on some drives, are only used for the optional rotational position sense feature.

On the A-cable, the Spare pair (59,60) is used by some Priam drives as an 11th bus bit. Fujitsu drives use this pair as well as bit 3 of the unit number as two extra tags for reading back drive status. The paddle board does not have to be aware of which drive type is being used, it only needs to allow the controller to transmit these signals.

On the A-cable, Dual Channel Busy is not used. Address Mark Found is used, not for that, but as one of the extra status signals from the Fujitsu drives.

Unit selection is done entirely by the drives. The unit number on the A cable is compared with unit switches in each drive, and the result comes back as Unit Selected on the B cable. The Unit Selected signals are used to gate the selected drive's B cable signals onto a bus on the paddle board. A PROM looks at the Unit Selected signals and provides a Select Error status unless one and only one is asserted. Unit Selected is terminated in such a way that missing or powered-down drives will not assert Unit Selected, even in the presence of noise (UNLIKE the case on the F-2!). 3.3K to +5 for the Unit Selected - signal, and 3.3K to -5 for the Unit Selected + signal; these form voltage dividers with the 56 ohm terminators to ground.

The sequencing signals, Pick and Hold, are both grounded by the paddle board when "Power OK" is true and otherwise left to float. These are TTL-compatible signals.

The Open Cable Detect or Channel Ready signal is a differential version of the same thing; note that it takes two 75110 sections to drive it.

The paddle board includes an ID prom (32x8) since it is a complex board. This prom can be read onto the disk bus, addressed by some of the tag lines, by a command from the controller. There is a diagnostic loopback feature which requires the paddle board to disconnect itself.

```
A-cable outputs:
Bus <10:0>, Tag<3:0>, Unit<3:0>         (19)
Open-Cable Detect
Sequence Pick
Sequence Hold
   (These are all 3 driven from Power OK)             Total of 22

A-cable inputs:
Status <5:0>, Sector, Index             (8)

B-cable outputs:
Write Data, Write Clock                 (2)

B-cable inputs:
Read Data, Read Clock, Servo Clock,
Index, Sector
Unit Selected                           (6)
 (Note that Unit Selected is not fed
   through to controller)

 Other Inputs:
 Select error - not exactly one drive selected

 Cabling to disk:  one 60-conductor, plus one 26-conductor per drive.
 Power requirements: +5, -5

 Interface circuits:
   [We are really using quad circuits I guess.]

   A-cable outputs: 1/2 75110, 56 ohms to ground
     except Open Cable Detect uses a whole 75110 and no termination
     except Sequence Pick, Hold: each 1/2 75452 and no termination
   B-cable outputs: 1/2 75110, no termination
   A-cable inputs: 1/2 75107B, 56 ohms to ground, 470 ohms in series
   B-cable inputs: 1/2 75108B, 82 ohms to ground, 470 ohms in series
                 All B-cable TTL data tied together with suitable pullups
                 thus implementing multiplexing
     except Index, Sector, Selected:
                   1/2 75107B, 56 ohms to ground, 470 ohms in series
   Select error - 32x8 Prom looking at Unit Selected lines
   ID prom - another 32x8 tristate prom
   Diagnostic disable - 74LS244 on signals listed below

 Total parts counts:  (assuming 4 B cables implemented)
   10  75110A each with 4 56-ohm resistors
    5  75110A
   10  75107B each with 4 56-ohm and 4 470-ohm resistors
         Also 8 3.3K resistors
    6  75108B each with 4 82-ohm and 4 470-ohm resistors
         Also 3 330-ohm(?) pullups for the open-collector mux
    2  32x8 prom  (74S288/TBP18S030)
    1  74LS244
   --
   34 active dips, 179 resistors (in sips and dips I assume)


 Priam Paddle Board:
```

Signal	n-bits	in/out	DIPs	Termination (paren if optional)
DBUS	8	IO	1 8304	(330/390)
AD	2	O	1/4 74S244?	(220/330 at drive)
-RD,-WR,-RESET	3	O	3/8 74S244?	(220/330 at drive)
-DRV SEL	4	O	1/2 74S244?	(220/330 at drive)
-HD SEL	3	O	3/8 74S373?	(220/330 at drive)
-RDY,-IDX,-SEC	3	I	3/8 74LS244	220/330
-WR GT,-RD GT	2	O	1/4 74S244?	(220/330 at drive)
WR DATA,WR CLK	2	O	1/2 26LS31	none
RD DATA,RD CLK	2	I	1/2 26LS32	100 from + to -

```
See the next page for how these are mapped into SMD signals,
making the two paddle boards compatible with the same controller.
The Priam does not use any radial cables, so the paddle board
handles any reasonable number of drives.  Priam does not provide any
way to detect multiple drive selection.

Cabling to disk: one 50-conductor
Power requirements: +5

Total parts counts:  (these are wrong)
  1 8304
  1 26LS31
  1 26LS32
  2 74LS244
  2 74S244
  1 74S373
  1 TBP18S030
  2 100 ohm resistors  (discrete?)
  8 330/390 resistors  (1 10-pin SIP?  2 8-pin SIPs?)
  3 220/330 resistors  (1 8-pin SIP)


Other stuff also on this paddle board:

TV: In the interim we will be using the A-machine TV interface.  This amounts
```

to 3 twisted pairs leaving the paddleboard. The ECL and TTL line drivers are on the paddle board, to make the paddle board be the field-replaceable-unit for line-driver blowouts. The keyboard input will go to the FEP paddleboard.

Eventually we will be using a new TV interface. Details are not yet worked out; this could be as many as 8 twisted pair and differential drivers for them.

NET: The cable to the ethernet transceiver is a 15-pin D connector. It carries three differential pairs, a shield, and about 1/2 amp of 12-volt power. It is not yet clear whether the line drivers and receivers will be on this board.
If so, parts requirements are: (if we follow the 3-com guidelines):

```
                3 10116
                approximately 50 resistors, diodes, and capacitors
```

Putting the line drivers and receivers on the paddleboard might necessitate putting the clock separator on this board also; it would be about the same size as the line drivers and receivers.
(On the 3COM board this is about 3"x9" of PC board total, I guess.)

The net has a separate diagnostic loopback feature, but shares the disk's paddle board ID prom.

Paddle Board Interface

Signal	SMD	Priam	
PADDLE BUS <1:0>	n/a	DBUS <1:0>	To paddle
PADDLE BUS <7:2>	BUS <7:2>	DBUS <7:2>	
PADDLE BUS <10:8>	BUS <10:8>	HEAD <2:0>	
PADDLE BUS <1:0> SMD	BUS <1:0>	n/a	
U READ GATE	n/a	READ GATE	
U WRITE GATE	n/a	WRITE GATE	
PADDLE UNIT <3:0>	UNIT <3:0>	UNIT <3:0>	
PADDLE TAG 0	UNIT TAG	-AD 0	
PADDLE TAG 1	CYLINDER TAG	-AD 1	
PADDLE TAG 2	HEAD TAG	WR	
PADDLE TAG 3	CONTROL TAG	RD	
PADDLE DBUS IN	n/a	(see below)	
PADDLE WRITE CLK	Write clock	n/a	
PADDLE WRITE DATA	Write data (all units)	Write data	
PADDLE POWER OK	Channel ready	-RESET	
	Sequence pick		
	Sequence hold		
PADDLE DISK ENABLE L	(see below)	(see below)	
PADDLE ID ENABLE L	(see below)	(see below)	
PADDLE INDEX	INDEX	INDEX	From paddle
PADDLE SECTOR	SECTOR	SECTOR	
PADDLE INDEX <0:3> L	Radial Index	n/a	
PADDLE SECTOR <0:3> L	Radial Sector	n/a	
PADDLE READ CLK	Read clock	Read/ref clock	
PADDLE SERVO CLK	Servo clock	Write clock	
PADDLE READ DATA	Read data (sel unit)	Read data	
PADDLE STATUS <5:0>	STATUS <5:0>	n/a	
PADDLE SELECT ERROR	SELECT ERROR	-READY	

Notes:

```
Total of 49 pins.  Network and TV also go through same paddleboard.
BUS <10> is also known as TAG <4> on Fujitsu SMD, spare on some SMD's
UNIT <3> is also known as TAG <5> on Fujitsu SMD
Radial index & sector only exist on some SMD's
PADDLE BUS <7:0> are tristate, for benefit of Priam (Lbus can read it back)
PADDLE DBUS IN if 1 enables Priam DBUS to PADDLE STATUS,
  if 0 enables PADDLE BUS to DBUS
SMD "B" (radial) cable inputs are multiplexed on the paddle board,
  using 75108's gated by unit-selected.  Pullup on paddle board.
PADDLE ID ENABLE L enables the ID prom to drive BUS <7:0>.  The prom
  address comes from TAG 1,UNIT<3:0>.  These bits can take on any combination
  without affecting the disk as long as the other tags are zero.
PADDLE DISK ENABLE L enables tristate drivers for SECTOR,
  INDEX, READ DATA, SERVO CLK, and READ CLK, enables the tag lines to
  be asserted to the disk, and enables SELECT ERROR (which is forced
  off if the disk is not enabled.)  Also if PADDLE DISK ENABLE is not
  asserted to a Priam paddle board, it disables BUS<7:0> from
  being driven by the disk (allowing it to be driven by the ID prom instead).
Clock phases:
  Read and write data both change on the falling edge of the
    corresponding clock that goes to/from the paddle board.
  For SMD, write clk is not inverted on paddle board, but
          read clk is.  Servo clk phase doesn't matter.
  For Priam, both clocks are inverted.  Actually a jumper is
    required on the paddle board for the phase of write clock,
    which must be set according to the cable length, since the
    write data are not accompanied by a clock from the controller.
Set switch on Priam for "drive originates write clock" mode
See Fujitsu manual or programming doc above for STATUS <5:0> mnemonics
SELECT ERROR comes from a prom addressed by
 the unit-selected lines from the 4 drives:
       SELECT ERROR is 0 if exactly one unit selected
       [The former "no select" signal has been flushed.]
SEEK END signal on SMD "B" cable is not used.
```

Possibility of Trident paddle board

Signal	SMD	Trident	
PADDLE BUS <1:0>	n/a	Bus <1:0>	To paddle
PADDLE BUS <9:2>	BUS <9:2>	Bus <9:2>	
PADDLE BUS <10>	BUS <10>	n/a	
PADDLE BUS <1:0> SMD	BUS <1:0>	n/a	
U READ GATE	n/a	READ GATE (see below)	
U WRITE GATE	n/a	WRITE GATE	
PADDLE UNIT <1:0>	UNIT <1:0>	UNIT <1:0>	
PADDLE UNIT <3:2>	UNIT <3:2>	(spare)	
PADDLE TAG 0	UNIT TAG	(spare)	
PADDLE TAG 1	CYLINDER TAG	CYLINDER TAG	
PADDLE TAG 2	HEAD TAG	HEAD TAG	
PADDLE TAG 3	CONTROL TAG	CONTROL TAG	
PADDLE DBUS IN	n/a	n/a	
PADDLE WRITE CLK	Write clock	n/a	
PADDLE WRITE DATA	Write data (all units)	Write data	
PADDLE POWER OK	Channel ready Sequence pick Sequence hold	Sequence	
PADDLE DISK ENABLE L		(same)	
PADDLE ID ENABLE L		(same)	
PADDLE INDEX	INDEX	INDEX	From paddle
PADDLE SECTOR	SECTOR	SECTOR	
PADDLE INDEX <0:3> L	Radial Index	n/a	
PADDLE SECTOR <0:3> L	Radial Sector	n/a	
PADDLE READ CLK	Read clock	Clock	
PADDLE SERVO CLK	Servo clock	same Clock	
PADDLE READ DATA	Read data (sel unit)	Read data	
PADDLE STATUS <0>	Ready	On-line	
PADDLE STATUS <1>	On Cylinder	Ready	
PADDLE STATUS <2>	Seek Error	Seek incomplete	
PADDLE STATUS <3>	Device Check	Device Check	
PADDLE STATUS <4>	Read Only	Read Only	
PADDLE STATUS <5>		n/a	
PADDLE SELECT ERROR	SELECT ERROR	SELECT ERROR	

Hard Issues:

Pre gate must be generated.
  Pre gate is turned on to enable the PLO to synchronize itself;
  Read gate is turned on later after it is known to be synchronized.
  When writing Pre Gate seems to be need to be turned on before
  Write Gate, but they can be turned off together.

There is only one clock; need phase synchronizer on paddle board.

One theory would be to sense the leading edge of READ GATE and start a delay. During the delay the clock is inhibited and pre gate is asserted. After the delay, read gate is asserted and the clock is permitted to restart on its next active edge. The state machine program will then need to be modified to take out the delays which the paddle board will supply itself.

Actually it appears from the totally cruddy T-300 prints that when the clock is switching from servo clock to read clock, it does not change phase abruptly; instead there is a single PLO which is sometimes synchronized to the servo track and sometimes to the data. I don't know whether we could rely on this.

The A machine controller holds pre gate and read gate stable while turning control tag on and off; is this really necessary? (I suspect this is only an artifact of that disk control's multiplexor that drives the disk bus; nothing in the T-300 seems to need it.)

Easy Issues:

Unit selection (decode unit number into select lines)
  Probably not worth bothering about checking for multiple units selected
Sequencing (could punt sequential power-up, blow circuit breakers)
  Alternatively could use some spare output as a sequence command
  and provide sequence latches for each drive on the paddle board

```
   (as in the DM board on the A machine).
No rotational position sensing unless we decode composite sector/index,
   which doesn't seem worth it.

Conclusion:

It appears to be possible.
```

```
Totally obsolete details of the state machine

Number of orders: 8 (?)
States per order: 32?
Outputs:
        Clock select (servo or read)  [processor or disk]
        Divide clock by: 1, 16, 36, others? (6 bit field)  [8 bit field]
        Read Gate
        Write Gate
        Data Connections Select: read, write, write-ecc, ecc-feedback
        Transfer data word to/from shift register from/to buffer register
         and wakeup microcode (effective at divided clock rate)
         [ECC is cleared by feeding it 0's for 32 clocks.]
        [Control tag (Busy)]
Sequencing controls:
        Loop condition:
                until sector pulse
                until disk read bit=1
                until stop request
                never
        Busy (versus stopped)
        Stop if ECC=0 (for Error Correct order)
        (Don't provide jump to other order, just copy Read after end
         of Read Header).

Guessed size of state machine PROM: 256 x 24
```

Note that when looping on stop-request is used, the clock division factor must be the same in the looping instruction and in the instruction that follows, because of instruction pipelining. No problem in practice. The other two looping conditions pay no attention to the clock division factor, hence don't have this problem.

Note that we assume that unlike the CADR DC, the state machine is fast enough to run at the disk bit rate. 10 Mhz is no problem, 20 MHz would be desirable. May want to use LS parts, to be replaced as necessary with ALS or S when 20 MHz disks come out. Also I'm assuming faster PALs will be available (perhaps at a premium) then.

Fujitsu track format

This is modified from the format used on Fujitsus by the A-machine SMD interface. Note that unlike the A machine, the L machine disk control does not complement the data. This format is valid for M2284 and M2312; it is not valid for M2351 (Eagle).

2 bits	Start-block synchronizer delay
16 bytes of zeros	Start-block gap (twice skew between data and servo heads)
11 bytes of zeros	VFO sync (data valid in 9 bytes maximum for M2284)
1 byte of 200	Sync pattern
4 bytes	Header (including redundancy check bits)
9 bytes of junk	Write splice (see below)
9 bytes of zeros	Rest of write splice
13 bytes of zeros	VFO resync
1 byte of 360	Sync pattern
1160 bytes	Data (64-bit tag, 256 36-bit words, or 288 32-bit words)
4 bytes	ECC
1 byte of zeros	Guard (for write turn-off)
11 bytes of zeros	Gap3
40 bytes of zeros	This sector's share of fragmentation (extra track length)

Total 1280 bytes per sector
Switches on VOFM board should be set to 1279 = #o 2377

Disk has 20480 bytes per track (20 pages unformatted exactly!)
16 sectors occupy this exactly

We could store 18 sectors of 256 32-bit words per track by changing the last gap length from 52 to 36. [Is this still true?]

Length of write splice issues:

It takes 4 bytes just to guarantee that the task has awakened and seen the header (longer than this and there would be an overrun). We then allow an additional 5 bytes (4 microseconds) for the microcode to clobber the state machine and get it started writing. The write splice actually has to be twice this length, because the write microcode has to write enough zeros to allow for it starting at the earliest possible time, while the read microcode has to delay long enough to allow for the write having started at the latest possible time. This length is also chosen to make things stay on word boundaries.

The extra 40 bytes of gap between sectors should give the microcode time to go out for a sandwich and a beer between sectors.

The following delays are built into the state machine program (version 12):

in slow-disk mode:

start of sector to read gate	8 bytes
start of write splice to read gate	9 bytes
read gate to read clock valid	10 bytes
start of write splice to write sync	22 bytes

in fast-disk mode:

start of sector to read gate	8 bytes
start of write splice to read gate	9 bytes
read gate to read clock valid	14 bytes
start of write splice to write sync	26 bytes

Format for M2351 (474 MB)

2 bits	Start-block synchronizer delay
16 bytes of zeros	Start-block gap (twice skew between data and servo heads)
16 bytes of zeros	VFO sync ("G1")
1 byte of 200	Sync pattern
4 bytes	Header (including redundancy check bits)
9 bytes of junk	Write splice
9 bytes of zeros	Rest of write splice
16 bytes of zeros	VFO resync

1 byte of 360	Sync pattern
1160 bytes	Data (64-bit tag, 256 36-bit words, or 288 32-bit words)
4 bytes	ECC
1 byte of zeros	Guard (for write turn-off)
15 bytes of zeros	Gap to meet 12 microsecond write-to-read delay (23 bytes)
28 bytes of zeros	This sector's share of fragmentation (extra track length)

Total 1280 bytes per sector
There are 28160 bytes per track, giving 22 sectors per track

The above format is designed to avoid having to change the state machine program.

Notes: the read gate to read clock valid time is not explicitly specified. It appears that it could be 7, 13, or 16 bytes. It is 9 bytes in the M2284. For now I am going to assume that 14 bytes will work. Or maybe the clock is always valid because the drive has an internal synchronizer?

No allowance for skew between data and servo heads is specified. I am assuming the same as on the M2284 (8 bytes maximum skew).

It is specified that there must be a minimum of 12 bytes from the start of write gate in the write splice to the sync bit, and a minimum of 16 bytes from the start of read gate there to the sync bit. This will be met.

Given the speed of the disk, it's unclear to me whether the 9-byte write splice is long enough.

```
State machine programs (for Fujitsu)

(The real thing is in <LMIOB>UDISK.LISP now)

Microinstruction fields:

  (next n) - defaults to tag of next microinstruction
  read-clock - defaults to servo-clock
  read-gate
  write-gate
  (wait cond) - read-data, start-block, or a number of bits
                3 or any even number ≤ 32 or any even number > 64 and ≤ 96.
  (data mode) - ecc-feedback, write-1  (the two mnemonics are synonymous)
                read
                write
                clear-ecc, write-ecc, write-0
                  (write-0 assuming ecc has been cleared for 32 bits)
  data-field
  advance - advance DMA task buffer or FEP buffer at end of word
  wakeup - wakeup DMA task at -start- of this state
  (func f) - err-if-start-block, stop-if-ecc=zero, done
  (test end-flag) - low bit of next state from end-flag (note: the bit
                is sampled as this state begins execution, not during its
                execution)
  (turnon fields...), (turnoff fields...) - set fields in this state
                and all following states (until turned off)
        (error if conflict between this and explicit fields)
  also there is hair for dispatching on fast-disk mode, see the code
```

```
Interface with rest of I/O board

        [Somewhat obsolete]

        Note: the network's interface will be quite similar, except for only being
        32 bits wide and probably having fewer memory-mapped I/O's.
        [If only life were simple]

Memory-mapped I/O decodes:

        READ DISK COMMAND L
        READ DISK ECC L
        READ DISK STATUS L
        READ DISK RPS L
        WRITE DISK COMMAND L
        WRITE DISK DIAG L
          Writes must be gated with the clock.

        It is allowed for the Lbus interface to use the same timing for memory
        mapped I/O as it uses for video buffer references.

Lbus microdevice decodes:
                (See earlier in the file for bit encodings)

        DISK DMA CYCLE
        DISK DISMISS L
        DISK TASK ACK
        READ DISK BUF L
        LB DEV 2 (i.e. end flag)

Lbus data path:

        LB DATA <35:0>
          Bidirectional data lines.  The Lbus interface takes care of
          deciding when to drive this onto the Lbus, independent of the
          disk interface deciding what data to drive onto LB DATA.
        DRIVE LBUS
          Signal from Lbus interface saying that LB DATA are output from
          this board.  Enables driving of fixnum tag onto LB DATA <35:32>
          except when the disk is driving those bits with 36-bit data.

Lbus controls:

        RESET L
        LB CLOCK
        LB CLOCK L
        LB WAIT L    (maybe this can be connected directly to bus?  Some DKTASK
                      logic can probably be merged into the common Lbus interface.)

Other Lbus signals connected directly to bus:

        LBUS POWER RESET L
        LBUS DEV COND L
        SPY DMA ENB L
        SPY DMA SYNC↑
        SPY DMA BUSY L
        SPY <7:0>
        TASK <15:8> REQ L
```

Notes on Wakeup timing advance

How many disk clocks should DISK WAKEUP precede DISK BUF ADVANCE by? Or in other words, how many disk clocks long should the state executing the femtoinstruction in which the WAKEUP and BUF ADVANCE bits are both set be? DISK WAKEUP sets simultaneously with the clocking of that femtoinstruction into the U register, and DISK BUF ADVANCE sets simultaneously with the clocking of the next femtoinstruction.

The minimum delay from DISK WAKEUP to DISK ACTIVE CYCLE, with all fast microinstructions, no traps, no bus interference, no higher priority task request, no task interference, no bus driving delays, and maximum good luck in the synchronizer is 4x180 = 720 ns. The relevant Lbus cycles are:

1. Synchronize task request
2. Disk in NEXT NEXT TASK
3. Disk in NEXT TASK
4. LBUS REQUEST
5. DISK ACTIVE CYCLE

There is a one disk clock delay from the setting of DISK BUF ADVANCE to the clocking of the buffer register that drives the Lbus. Thus for reading from disk, writing into memory, the delays are:

Disk	max Clock Period		# periods in 720 ns minus one	Actual femtocode
M2284	129.8	(min 117.4)	4.546	4
M2312	103.7	(min 99.7)	5.943	4
M2351	69.24	(min 65.24)	9.39	10
T306	107.0	(min 99.8)	5.728	4

We are cheating by 42 ns on the M2351, but the clock-to-output delay of the 74LS646 is 35 ns, compared with 75 ns from Lbus clock to output (74F175 then 74LS646 turnon delay). Also DISK WAKEUP doesn't get to the SQ board instantaneously.

For reading from memory and writing onto disk, there are two additional Lbus clocks before the disk buffer gets clocked. The shift register is clocked from the buffer register simultaneously with the setting of DISK BUF ADVANCE, so there is no additional one disk clock delay.

Disk	max Clock Period		# periods in 1080 ns	Actual femtocode
M2284	129.8	(min 117.4)	8.32	8
M2312	103.7	(min 99.7)	10.41	8
M2351	69.24	(min 65.24)	15.59	14
T306	107.0	(min 99.8)	10.09	8

Note that if we ever hook up a disk with a bit rate even slower than the M2284, we will need to provide special femtocode proms for it.

```
>> Notes:
1) Leave room for edge around displayed bit map.
2) We ordered:
        HD17H landscape, P40 phosphor, 36 kHz horizontal, 42 Hz vertical
        HD17V portrait, PC105 phosphor, 36 kHz horizontal, 74 Hz vertical
          (actually came with P39 phosphor)
        VR-1000 landscape, P104 phosphor
3) Question: are we going to lose do to 60 Hz hum?

>> Summary:

Star format:
  Landscape format, 17" CRT, visible area 10.6" x 13.6"
  Video rate 47.5 Mbit
  Horiz line rate 35 kHz        ; (7 usec blanking)
  Vert frame rate 38.7 Hz interlaced
  H/V dimensions 1024 x 809
  P40 phosphor

Ball HD-17 format:
  (Horiz blanking time 7.0 usec)
  (Max horiz line rate 38 kHz)
  (Vert blanking time 500 usec)

  1040 x 934 (interlaced)
  1040 x 881 (non-interlaced)
  1486 x 934 (fast bit time)
  1486 x 881

;H's "perfect" formats for Ball HD17
Landscape:
                                                                (Star)
        80 Hz/40 Hz vert        60/30           80 Hz/40 Hz     77.4 Hz/38.7 Hz
        50 Mhz video            50 Mhz          50 Mhz video    47.5 Mhz
        34.4 kHz horiz          30.24           36 kHz          35 kHz
        1102 x 826 raster       1304 x 978      1040 x 864      1024 x 809
        1.33 x 1 aspect         1.33 x 1        1.20 x 1        1.26 x 1

;;various other possible formats --  Landscape, interlaced, P40 phosphor
;; all assume 7 usec Hblank, 500 usec Vblank

        50 Mhz  37/74 Hz vert   33.3 kHz        1152 x 866
        50 Mhz  40/80 Hz vert   34.8 kHz        1088 x 834
        50 Mhz  42/84 Hz vert   35.5 kHz        1056 x 810


Portrait:
        80/40 Hz vert           37 Hz           34 Hz           80 Hz/40 Hz
        36 Mhz video            50 Mhz          37 Mhz          50 Mhz
        36 kHz horiz            36 kHz          36 kHz          43.8 kHz (?)
        768 x 924               1040 x 937      768 x 1024      789 x 1052
        1 x 1.20                1 x 0.90        1 x 1.33        1 x 1.33

Moniterm VR-1000
  Portrait format, 17" CRT
  Horiz line rate 64 kHz
  Vert frame rate 60 Hz
  Video rate 64 Mhz
  H/V dimensions 793 x 954
  (Horiz blanking time 3.5 usec)
  (Vert blanking time 700 usec)

Philips recommended format:
  Landscape format, 20" CRT, visible area 14" x 9"
  51 kHz Scanning rate
  Vert frame rate 60 Hz, non-interlaced
  Video rate 87 Mhz
  H/V dimensions 1280 x 809,  32K x 32B raster memory
  (Horiz blanking time 5.0 usec)
  (Max horiz line rate 51 kHz)
  (Vert blanking time 700 usec)

Philips ultimate format:
  Landscape format, 20" monitor, visible area 14" x 9"
  Horiz line rate 56 kHz
  Vert frame rate 60 Hz, non-interlaced
  Video rate 110 Mhz
  H/V dimensions 1400 x 900,  39K x 32b raster memory

Philips format limited to 64 Mhz video rate:
  Landscape format, 20" CRT, visible area 14" x 9"
  Horiz line rate 50 kHz
  Vert frame rate 60 Hz, non-interlaced
  Video rate 64 Mhz
  H/V dimensions 1024 x 800,  25K x 32B raster memory
```

```
Color Monitors:

>> Hitatchi C-3619

Horizontal: 31 KHz rate

  32.3 Usec Horizontal time
   6.0 Usec Horizontal blanking time
  --------------------------------
  26.3 Usec Horizontal data time

  25 Ns         1050 Bits horizontal

Portrait: 60 Hz rate (interlaced)

  16.7 Msec Vertical time
    .6 Msec vertical blanking time
  --------------------------------
  16.1 Msec Vertical data time

  467 Lines/Field

  1024 Lines

>>Hitachi C-3910 19" High Resolution 3-gun color display (random?)

Horizontal rate:        15-18 kHz
Vertical Rate:          40-60 Hz
Effective screen size:  394(W) x 295(H) mm
Effective triads:       1270 W x 951 H
Video bandwidth:        50 Hz - 25 Mhz
Video Rise/Fall times:  better than 20 ns
Convergence:            better than 0.5 mm center circle
Dynamic convergence:    10 "districts"
Dynamic focus:          yes
Horizontal blanking:    9 usecs
Vertical Blanking:      600 usecs (3H vert sync, 15-19H topmar, 3H botmar)

>>Hitachi C-6512 25" High Resolution 3-gun color display (random?)

Horizontal rate:        28-34 kHz
Vertical Rate:          40-70 Hz
Effective screen size:  480(W) x 350(H) mm
Effective triads:       1548 W x 1129 H
Video bandwidth:        50 Hz - 40 Mhz
Video Rise/Fall times:  better than 11 ns
Convergence:            better than 0.65 mm center circle
Dynamic convergence:    10 "districts"
Dynamic focus:          yes
Horizontal blanking:    6.5 usecs (2.75 usec Hsync, 7.0 usec topmar, 0.8 botmar)
Vertical Blanking:      600 usecs (3H vert sync, 15-19H topmar, 3H botmar)

>>Hitachi C-6912 19" High Resolution 3-gun color display

Horizontal rate:        28-35 kHz
Vertical Rate:          50-60 Hz
Effective screen size:  394(W) x 295(H) mm
Effective triads:       1270 W x 951 H
Video bandwidth:        50 Hz - 43 Mhz
Video Rise/Fall times:  better than 11 ns
Convergence:            better than 0.5 mm center circle
Dynamic convergence:    10 "districts"
Dynamic focus:          yes
Horizontal blanking:    6.0 usecs (2.8 usec Hsync, 7.5 topmar, 0.8 botmar)
Vertical Blanking:      600 usecs (3H vert sync, 20-29H topmar, 7H botmar)

>>Hitachi C-6919 19" High Resolution inline-gun color display

Horizontal rate:        28-35 kHz
Vertical Rate:          40-70 Hz
Effective screen size:  350(W) x 270(H) mm
Effective triads:       1129 W x 870 H
Video bandwidth:        50 Hz - 40 Mhz
Video Rise/Fall times:  better than 11 ns
Convergence:            better than 0.75 mm center circle
```

Dynamic convergence:	10 "districts"
Dynamic focus:	yes
Horizontal blanking:	6.0 usecs (2.8 usec Hsync, 6.0 topmar, 0.7 botmar)
Vertical Blanking:	700 usecs (3H vert sync, ?? topmar, 3H botmar)

Formats for 6912

1) Landscape, interlaced

```
        Interlaced 30/60
        1248 x 950 pels   (1.31)
        28.5 kHz horizontal
        43 Mhz video
```

>> Hitachi C-8912 19" High resolution

Horizontal rate:	37-45 kHz
Vertical Rate:	40-70 Hz
Effective screen size:	394(W) x 295(H) mm
Effective triads:	1270 W x 951 H
Video bandwidth:	50 Hz - 55 Mhz
Video Rise/Fall times:	better than 8/7 ns
Convergence:	better than 0.5 mm center circle
Dynamic convergence:	10 "districts"
Dynamic focus:	yes
Horizontal blanking:	6 usecs
Vertical Blanking:	600 usecs (3H vert sync, 26H topmar, 3H botmar)

Formats for C-8912

1) Landscape.

```
        Interlaced 30/60.
        1280W x 1024H pels  (W/H = 1.25)
        30.6 kHz Horizontal
        48 Mhz video

        (Interlaced 33/66)
        1280W x 1024H   (W/H = 1.25)
        34.1 kHz Horizontal
        55 Mhz video

        (requires 196 64K RAMS!)
        1408 x 1054 pels  (W/H = 1.333)
        31.6 kHz horizontal
        55 Mhz video
```

These are useful for image display and graphics where you can stand the flicker (draw lines at least 2 pels wide.) You can also use the slow phosphor maybe.

2) Landscape, non-interlaced 60 Hz.

```
        960W x 720H pels  (W/H = 1.33)
        43.2 kHz horizontal
        56 Mhz video            ; slightly fast
```

1:38pm Thursday, 12 November 1981

Called VMI, 715-834-7785
Mr Kveberg

Asked about Landscape format 17" monitor. They mostly make 15 and 20" in horizontal format, may have to order 17" tube special from Clinton. With 20", the line spacing is 12.5 mils, and the line width is about 10 mils, so the raster may be visible in the center of the screen.

They have 3 types of anti-reflection coatings:

OCLI
Etched (Panasonic etched is best, only in 15" now)
Dataview coating (made by Clinton, unclear what this is)

Stator yoke, basically improves corner focus at the cost of higher power dissipation. They are working on reducing the power requirements, and may go to stator yokes on the standard model. Some of the saddle yokes can't be corner focused, and have to be pulled off the production line (maybe PERQ is ordering saddle yokes).

They have 2 dynamic focus options, 350V and 700V. Use the 700V on the 20" tube. Perhaps the 17" can get by with 350V, although the dynamic focus on our current 15" monitors doesn't seem to be particularly effective.

Will call back with quotes and availability Fri or Monday.

Horizontal format would be the M17L800H. This gives 1024 x 800 with 50 kHz scanning, and 64 Mhz video.

```
10:29pm  Tuesday, 17 November 1981

Sent letter to Philips begging them for display.

Correct ultimate format:

        Landscape 20" monitor, giving 14" x 9" visible area
        Scan rate 56 kHz
        Refresh rate 60 Hz
        Video rate 110 Mhz
        Resolution 100 pts/inch, or 1400 x 900, approx 39K x 32b
```

10:34pm Tuesday, 17 November 1981
Bits about the PERQ monitors.

They use saddle yokes. This probably explains why they reject some percentage of their incoming yokes. They seem to add their own magnets to trim the yoke.

The "extra DIP" in the video circuitry is a 74S32, which is probably used to differentially stretch the rising or falling edge of the video to optimize single dots on the screen. I assume they are feeding the monitor with TTL video.

They use the same tube, Clinton CE675M15PC104TE.

Design Notes for the physical page tag hardware (revised 3/21/82)

[This file supersedes the file GC-PAGE-TAG.DESIGN]

There is an entry for each physical page of main memory. To save space on the board, only the bottom half of physical address space (16 memory boards) is tagged. This should not be a problem since we won't be supporting backplanes larger than this. By the time we are ready for more physical address space we can switch to 64K static rams.

Each entry consists of two tag bits and a parity check bit. One bit is set whenever this physical page is read or written as a virtual page; it is called the Referenced bit. The second bit is set whenever a pointer to a "controlled" space is written into this physical page; it is called the GC tag bit. The page tags see only "normal" memory references. They do not see references by the FEP, DMA references, or references with SPEC INHIBIT PAGE TAGS asserted. They do see all memory references addressed through the VMA (with the VMA containing either a virtual or a physical address), and most non-DMA references by I/O tasks. This is done to avoid spuriously setting the GC tag bit when the write data does not come from the data path and to avoid spuriously setting the Referenced bit when scanning main memory for correctable errors.

The Referenced bit is used by the paging system to detect whether pages are good candidates for removal from main memory. The advantage of putting this in the page tag hardware is that the overhead for "age traps" can be avoided while still making it possible to monitor the use of pages. Detection of modification of pages will continue to be done with "read-write first" trapping; the overhead for this is one page-map miss per disk write, which is negligible.

The GC tag bit records pages which contain pointers to "temporary" (short lifetime) areas. When such an area is to be reclaimed, a scan through the tag bits will quickly locate the presumably small set of pages containing such pointers. A scan through those pages allows the pointed-to objects to be evacuated, and the pointers to be relocated. This avoids the need to scan the entire virtual memory when reclaiming (a region of) a short-lifetime area. A separate table, maintained by software, remembers which disk pages contain pointers to which controlled areas.

In addition to the automatic setting of tag bits, the microcode may set and clear the tag bits, and may read them one at a time as skip conditions. It is possible to make a microcode loop which scans the tag bits at the rate of two microinstruction cycles per page. This amounts to 1 millisecond per 750K of main memory, which is acceptably fast (other parts of the GC operation still dominate it).

When a disk page is read into a main memory page, and after a GC cycle, the microcode sets the GC tag bit to the appropriate value (could be either 0 or 1). When the garbage collector wants to reclaim some temporary space, it scans the bits for all pages of main memory being used for paging; for each page with a 1 bit, it scans all words in that page, looking for pointers to condemned-temporary-space; for each such pointer it copies out the object pointed-to and adjusts the pointer. Non-hardware tables are used to determine whether any pages containing pointers to the condemned temporary space are on the disk; this is aided by checking the page's GC tag bit when a page is written to disk; if the bit is set the page must be scanned to determine whether it really points to temporary space, and which temporary spaces are pointed to. This scan can be overlapped with the disk write of course.

Implementation

Originally the page tag bits were going to be in 64K dynamic RAM, however the timing requirements proved to be impractical. Hence they are made out of 16K static RAM, which is why only the bottom 16 slot numbers may be used for memory boards.

The following inputs exist:

```
LBUS ADDR <23:19> - the physical page to be accessed next.
NORMAL ACTIVE L - true if this is an active cycle and the page tags are
                supposed to see it.
LBUS STATE CLK L - the clock gated by LBUS WAIT.
DP SET GC TAG L - true during an active cycle if the datapath output during the
                previous cycle was a pointer and its address was in a temporary
                space.  If this active cycle is for a virtual write, the GC
                tag bit needs to be set.
WRITE ACTIVE L - true during an active write cycle (registered version of
                LBUS WRITE L).
WRITE PAGE TAG L - true if lbus-dev-write of the page tag being done
READ PAGE TAG L - true if reading page tag (via lbus-dev-write!)
LBUS DEV <4:3> - modifiers for the above
                [Note: the spec and magic fields could be used instead of
                the microdevice I/O.  This probably depends on whether the
                page tags are housed on the MC board or the FEP board.
                Use of the spec field would require some care since it also
                must be used to inhibit task switching.]
```

The following outputs exist:

```
LBUS DEV COND L - asserted when READ PAGE TAG and the selected tag bit is set.
PAGE TAG PAR ERR L - asserted when bad parity is read from the page tags.
```

Microcode control:

One selects a physical page by doing a read of any location in the page. Normally the address would be supplied as a physical address on the Abus although the VMA could also be used. Actually starting a read isn't necessary; it's only necessary to convince the memory control to put the physical address on the Lbus; however, starting a read is the only way to do this. In the next cycle one uses a microdevice operation to read or write the page tags for the addressed page.

Since the address is supplied in the previous cycle before the read and write, it is necessary to prevent a task switch from intervening. By starting a memory read in order to emit the address, this task interlock is taken care of automatically. The memory control causes the microcode to wait if it tries to start a memory operation when a task switch is about to happen. Therefore the microdevice operation will always follow immediately upon the heels of the start-read. A problem could still occur if the microdevice operation has to wait (MC WAIT), because it is possible to switch tasks during such a wait, and the page tag address would change in the interim. MC WAIT cannot be caused by the bus being busy, because the bus cannot be busy during a non-block read active cycle (study the bus control logic). Memory-interleaving rules ensure that a memory cycle cannot be started by the FEP or the IFU at a time that would cause the bus to be busy during this cycle. MC WAIT cannot be caused by LBUS WAIT, by simple fiat:

******* NOTE WELL *******
Main memory, of the kind that is used to hold virtual pages, cannot use the LBUS WAIT feature. If main memory ever drives LBUS WAIT, the page tags will stop working.

The page tag's address register is not clocked when LBUS WAIT is asserted. The only case where this matters is for normal accesses to boards that assert LBUS WAIT, such as TV memory; this prevents the address from changing and the wrong page tag being written. Of course page tags for TV memory are meaningless, but it is important not to bash page tags for main memory when accessing TV memory.

WRITE PAGE TAG L is asserted during second half when writing to microdevice slot 36, subdevice 1 (on the FEP board).
LBUS DEV 3 is written into the selected bit. The other remains unchanged.

```
LBUS DEV 4 selects which bit:
        0 the gc tag bit
        1 the referenced bit

READ PAGE TAG L is asserted when writing to microdevice slot 36 subdevice 3.
LBUS DEV <4:3> select the bit to read, as follows:
        00 - the gc tag bit
        01 - the referenced bit
        10 - the parity bit
        11 - (not used)
The selected bit comes back on the LBUS DEV COND L line and may be used as
a skip condition.

Scanning GC page tags:

We simplify the hardware by scanning at the slow rate of 2 cycles per
bit.  This amounts to 1 millisecond per 750K of main memory, which is
acceptably fast (other parts of the GC operation still dominate it).
The microcode alternates between cycles which emit a physical address
on the Abus, start a read, and do a compare to check for being done, and
cycles which increment the physical address and also skip on the tag bit,
into either the first cycle again or the start of the word scanning loop.

There is no special function for writing a pointer into main memory to
enable the check and setting of gc page tag.  Instead, any write into main
memory at a virtual address, where the data type map says the type is a
pointer, and the gc map says it points at temporary space, will set the
addressed gc page tag bit in the following cycle if necessary.
```

```
eived: from SCRC-VALLECITO.ARPA by GODZILLA.SCH.Symbolics.COM via CHAOS with CHAOS-MAIL id 166949; Wed 1
2-Feb-86 12:23:25-PST
Received: from SCRC-STONY-BROOK.ARPA by SCRC-VALLECITO.ARPA via CHAOS with CHAOS-MAIL id 84626; Wed 12-F
eb-86 15:22:30-EST
Received: from GOSHAWK.SCRC.Symbolics.COM by SCRC-STONY-BROOK.ARPA via CHAOS with CHAOS-MAIL id 415782;
Wed 12-Feb-86 15:21:09-EST
Date: Wed, 12 Feb 86 15:17 EST
From: Robin L. Anderson <ANDERSON@SCRC-VALLECITO.ARPA>
Subject: I plan
To: dcook@CUPID.SCRC.Symbolics.COM, yang@SCRC-VALLECITO.ARPA
cc: anderson@CUPID.SCRC.Symbolics.COM
In-Reply-To: The message of 12 Feb 86 11:47-EST from Dale Cook <dcook@CUPID.SCRC.Symbolics.COM>
Message-ID: <860212151726.3.ANDERSON@GOSHAWK.SCRC.Symbolics.COM>

    Received: from SCRC-STONY-BROOK.ARPA by SCRC-VALLECITO.ARPA via CHAOS with CHAOS-MAIL id 84488; Wed
12-Feb-86 11:54:25-EST
    Received: from CUPID.SCRC.Symbolics.COM by SCRC-STONY-BROOK.ARPA via CHAOS with CHAOS-MAIL id 415503
; Wed 12-Feb-86 11:51:20-EST
    Received: by scrc-cupid id AA10401; Wed, 12 Feb 86 11:47:31 est
    Date: Wed, 12 Feb 86 11:47:31 est
    From: Dale Cook <dcook@cupid>
    To: jevans@cupid, schnorr@cupid
    Subject: I plan
    Cc: anderson@cupid, bellomy@cupid, dcook@cupid
```

> From our (diagnostic's) point of view there seems to be an enabling document missing. Traditionally, we are the tail of the dog on any project, for mostly good reason (we need more of the plans and specifications firmed up before we can start) and are, in some senses, a "service" to other groups. In that light, Services and Manufacturing should probably put together Requirements documents stating: a) testability goals to meet their cost goals, b) diagnostic goals (test time, coverage, etc) to meet goals and c) description of any pieces they intend to implement themselves (SCH Diagnostics, Doug Evans etc.) We can then "respond" to those requirements. Lord knows, I'm no fan of paper and don't want to be responsible for creating any additional. Glen has informally agreed to provide me input by EMAIL. Perhaps what is needed is a similar arrangement with Chatsworth. Is Aicher the guy or is there some one more technical I could begin a dialogue with? What's needed is someone who can help trade off life cycle and production costs against hardware burden - we can make this beast arbitrarily diagnosable, but only at arbitrarily high cost. The trick is bang for buck. (We could do this ourselves based on past experience, but it is better if our (diagnostic's) customers participate so they know up front why we did what we did. [Sorry for the length of this; got carried away.]
> - Dale

Sam, this was sent around here to new product people. I thought you might have some input from your group - what requirements you have for diagnostics and what equipment you have available for testing new products, etc. If you have any thoughts, send them on to us.

I'll be in Westwood next week (feb. 18-21). Probably won't get to Chatsworth unless there is some pressing need there. We are doing a short G machine study with Ron Lebel's group. IF you need anything that I can bring to the West Coast, let me know.
-Robin

From: Robin L. Anderson <ANDERSON@SCRC-VALLECITO.ARPA>
Subject: gmachine info references
To: yang@SCRC-VALLECITO.ARPA
cc: dcook@SCRC-VALLECITO.ARPA, anderson@SCRC-VALLECITO.ARPA
Message-ID: <860213174849.7.ANDERSON@GOSHAWK.SCRC.Symbolics.COM>

Sam, I got a phone call from Jahan. He wants references for the L machine old FEP. I found a directory he should read:

```
        vallecito:>hardware>lmhard>
```

There is a file called "dma.design." He will want to read it. The entire directory might be of interest to him. IT contains mucho info on the L machine. Will you pass this message on to him? He really needs to get onto the system somehow. Thanks.

The G machine information is in:

```
        white:>gmach>gmach>
        white:>gawboy>
```

and environs. Look all around in lpalmer's directory and all over on white. The text files are good documents about how the machine works and things.

Vallecito is in Cambridge and we call it "v". White is in Westwood. Both are open to you from Godzilla.

I hope this helps. You are still missing the schematics themselves. They are not available on the system, since they were developed on the Mentor System at Westwood. Heidi can print copies of the schematics for you there. I will get some for you while I am there next week. If I cannot get time to drive to Chatsworth, I will mail them. OK?

See you later,
Robin

To : Robin ANDerson
FROM - Jahan
Date 2-14-86
Subject reply

Thank you FOR ALL The information
It ~~was~~ is very helpfull.

Lucifer Disk Bulkhead design

Bulkhead needs to provide connectors for:

Net

Net electrical interface will be on the main board (ECL drive/receive). Power will be provided through a paddle board fuse directly to the cable from the backplane.

Total pinout is 9 wires on a 15 pin type D female connector. This is three pairs (one transmit, two receive) of ECL differential, plus power, ground. The transceiver power is speced at +12 to +15 volts +/- 5% at .5 amps.

Total 6 wires to main board. The receiver is relatively hairy, and probably doesn't fit on the paddleboard. It has ugly discrete components.

We need to pick up chasis ground for the cable shield through some path. This is also the case for shields of other cables. Probably the best way is through spliting off of the ground pins for the flat cable at the connector, and to rely on grounding of the metal shells of the connectors.

Console

Interface logic will be on main board. No special logic or components on paddleboard

Signals consist of seven pairs terminated with TTL line drivers/receivers. These signals are:

```
transmit:
        Data bus 0:3
        Clock
        Tag
receiver:
        Input
```

These could be located on the paddleboard, or they could be done on the main board.

total 7 signals if received/transmitted on paddleboard,
total 14 signals if received/transmitted from main board.

Disk

A cable drivers for the following signals:

```
Tag 0:5
Bus 0:9
Channel Ready   (wants double driver and no pulldowns)
Unit Select 0:2 (tag<5> is sometimes named unit select<3>)
        total 20 signals
```

A cable receivers for the following signals:

```
Status 0:5
Sector
Index
        total 8 signals
```

<< unclear treatment of pick and hold >>

B cable receivers for the following signals:

```
Servo write clock  -- gated by unit selected
Read clock  -- gated by unit selected
Read Data  -- gated by unit selected

Unit Selected   -- ungated, driven to controller separately
                [may go into a prom and just SELECT ERROR to controller]
Seek End        -- ungated, driven to controller separately
                [seek end not being used at all -- Moon]
        total 5 signals
```

B cable drivers for the following signals:

```
- Write clock  -- inverted to make life easier for the controller
Write data
These are ungated and driven to all drives simultaneously

        total 2 signals

Note:  B cable receivers are open collector for Servo clock, read clock, and read data.
Unit Selected and seek end are always
enabled.  The Unit Selected enables the open collector B cable
receivers.  It is passed onto the main board for detection of multiple
select [and selected unit number--no] (probably this is done with a PROM).
[The seek end signal is passed onto the controller separately for each drive.--no]

B cable drivers are always enabled.


Terminations:


A cable:

drivers:  56 ohms to ground
receivers:      56 ohms to ground
                470 ohms series


B cable:

drivers:  No terminations
receivers:      82 ohms pulldown to ground
                        (except 56 ohms for seek end, unit selected)
                470 ohm series

Total signals to main board from disk logic:

A cable transmit:                                  20
A cable receive:                                   8
B cable transmit:                                  2
B cable receive (not incl. unit selected,
                or seek end):                      3

(assume four units)
per unit (B cable unit selected & seek end)        8  [4 or 0]
                                               ----------
total pinout to main board from disk:              41 pins

Grand total pinout needed to main board:

Net:            6 (possibly 3)
TV:             14 (possibly 7)
Disk:           38 [fewer]
Other disk:     6-8? (pins used by Priam paddleboard but not SMD paddle)
         ------------
                61 pins [fewer]


Voltages needed on the paddleboard:

+5, ground, -5, +12, chassis ground


Other issues: pinout compatibility with Priam interface?
                 [this has been taken care of--Moon]
              pinout compatibility with Chaosnet transceiver?
              FEP ability to sense blown fuses?
```

```
This is the layout of the current microword:                    -*-Text-*-

The first line of each description gives bit numbers in the control-memory
(see the CMMFLD print), signal names, and a very brief description.
Following that the detailed meaning of the microinstruction field, and
meanings of the decoded values where appropriate, are given.
When the same bits are used for more than one thing, they are listed
twice in succession.


This is accurate as of 12/17/81.
Updated 7/21/82 for TMC.
Updated 5/29/84 to be current, but not guaranteed to have been
updated and checked comprehensively.  Probably not up to date for IFU.
5/29/84 removed the documentation for "temporary mem control" on FEP rev-1.
6/27/84 added documentation for the IFU
4/24/85 changed MC Abus source addressing so that U AMRA <9> is 1, it was
previously a don't care. This is part of IFU/FPA compatability fix (Chas).

11-0            U AMRA <11:0>           A memory read address
        <11:0> can be an immediate A-memory address
        <7:0> can be an offset from a base register
        <8> can select the offset from the macroinstruction instead
        <10:9> can be U R BASE.
        <8:0> can select Abus sources on the MC board.
        <4:0> are the address for the board-ID prom.

10-9            U R BASE <1:0>          A memory base register select
        Same as U AMRA <10:9>.  Base registers are:
                0  Stack Pointer
                1  Frame Pointer
                2  Extra Base
                3  INST<7>=1 means Stack Pointer, =0 means Frame Pointer
                   Also adds 1 if INST<7>=1, thus 177 offset means zero.

13-12           U AMRA SEL <1:0>        Selects interpretation of U AMRA
        This field controls the A-memory read address and the Abus source.

        0  U AMRA <11:0> is an immediate address.
        1  LBUS ADDR <11:0> is the A-memory address.  Abus source is main
           memory unless the bus address lies within A-memory. U AMRA<8:6>
           should be zero and U R BASE<1:0> should be 3.
        2  Base register plus offset, within 1K bank selected by STACK BASE.
           U R BASE <1:0> selects the base, U AMRA <8:0> selects the offset.
        3  Abus source is not A-memory.  If U R BASE <1:0> is 0 or 1, the
           Abus source is the Stack Pointer or the Frame Pointer (28
           bits). If U R BASE <1:0> is 3 the Abus source is something on
           the MC board selected by U AMRA <8:6>:
             For TMC5:
                0 memory (MD)
                1 Lbus device (passes through MD)
                2 VMA  (reads ASN in bits <35:28> except for IFU)
                3 map  (for diagnostics, see below)
                4 PC   (a word address and dtp-even-pc or dtp-odd-pc in the type field)

             For IFU:
                0 memory-data (MD)
                1 Other memory-data (OTHER-MD). This is used for
                        microdevice reads.
                2 VMA (must use U MEM = Reserve)
                3 EPC (a word address and dtp-even-pc or dtp-odd-pc in the type field)
                4 memory-data-advance (Read MD, and increment the VMA.
                        Change the selected MD to be the other MD)

                5 PHTA-ASN/IIR If #0=0 then read the PHTA-ASN register.
                        If #0=1, then read the IIR (for debugging).
                        Must use U MEM = Reserve
                6 MAP Read back map contents from selected map (U MEM = Reserve)
                7 Map Read back map selected by #0 (U MEM = Reserve)

       Note: the command to increment the INST register (immediate operand to
       current instruction) is U AMRA SEL = 2, U AMRA<8:7> = 3.  In other words,
       reading base+offset, with offset coming from INST, and an extra bit
       turned on.  This satisfies the requirements of the array-register microcode.

14              U XYBUS SEL             X & Y Bus select
```

```
         0  X=A, Y=B
         1  X=B, Y=A
         (Modified by the Multiply and Crocks to Ybus special functions)

15               U STKP COUNT             Stack Pointer Count
         A 1 enables the stack pointer to count.  It increments if bit 11 of
         the A memory write address field is 1, decrements if it is 0.

27-16            U AMWA <11:0>            A memory write address
         <11:0> can be an immediate A-memory address
         <7:0> can be an offset from a base register
         <8> can select the offset from the macroinstruction instead
         <10:9> can be U W BASE
         <9:0> can be LBUS DEV <9:0>
         <10> can be memory write enable
         <11> can be stack-pointer count direction
         <11> selects which crock feeds Ybus
         <7:4> can be an extension of the U BMWA field
         <4:0> can be a byte rotation
         <9:5> can be a byte size

25-16            LBUS DEV <9:0>           Lbus device address
         Same as U AMWA <9:0>
         Bits <9:5> select a board and bits <4:0> select a device and
         register within that board.  This is for microdevice commands,
         including operation of DMA devices by their micro tasks.

26-25            U W BASE <1:0>           A memory base register select
         Same as U AMWA <10:9>.  Base registers are:
                 0  Stack Pointer
                 1  Frame Pointer
                 2  Extra Base
                 3  INST<7>=1 means Stack Pointer, =0 means Frame Pointer
                    Also adds 1 if INST<7>=1, thus 177 offset means zero.

27               --                       Stack-pointer count direction
         Same as U AMWA <11>
         1 if the stack-pointer is to increment, 0 if it is to decrement.
         Only takes effect if U STKP COUNT is on.

29-28            U AMWA SEL <1:0>         Selects interpretation of U AMWA
         This field controls the A-memory write address.

         0  U AMWA <11:0> is an immediate address.
         1  Base register plus offset, within 1K bank selected by STACK BASE.
            U W BASE <1:0> selects the base, U AMWA <8:0> selects the offset.
         2  The A-memory write address is the same as the read address.
            This is mainly useful when the U AMWA field is busy being the
            Lbus device address or the byte rotation and size.
         3  LBUS ADDR <11:0> is the A-memory address.  However, the write
            into A memory is suppressed unless U AMWA <10> = 1 and the bus
            address lies within A-memory.
            U AMWA <9:0> are the microdevice address, with special values
            write registers on the MC board (and the map).  The special values use slot
            number 37 in the device address.
            The TMC board doesn't look at U AMWA SEL nor at U AMWA <10>, however
            these are used to tell the DP a write into main memory or microdevice
            write is being done instead of an A-mem write.

31-30            U SEQ <1:0>              Sequencer Function
         0  No special operation.
         1  Pushj.  The control-stack pointer is incremented.
         2  Dismiss.  The current task is dismissed.  Tasks 1-7 run two
            more instructions, tasks 8-15 run one more instruction.
            (That's inaccurate; see the microcode manual.)
         3  Popj.  The control-stack pointer is decremented.

39-32            U BMRA <7:0>             B memory read address
         Controls the Bbus source.
         Locations 360-377 are the normal scratchpad locations.
         Locations 10-357 are constants (require special function to write).
         Location 0 (also 1-3) are unsigned immediate macroinstruction.
         Location 4 (also 5-7) are signed immediate macroinstruction.
         The immediate data come from INST <7:0>.  INST <7> is the sign bit.

43-40            U BMWA <3:0>             B memory write address
```

```
        Normally locations 360-377 are written.  Some location is always
        written unless the instruction is NOPed.  The special function
        Extended BMWA takes bits 7-4 of the address from U AMWA <7:4>.

44              U BMEM FROM XBUS        B memory write data select
        0  Write data comes from OBUS<35:0>
        1  Write data comes from ABUS<35:32>|XBUS<31:0>

47-45           U MEM <2:0>                 Memory control function
        In addition to this field, the memory control is also controlled
        by SPEC, AMRA, and AMWA.  This field just exists for the operations
        which need to be done in parallel with other things.

        The following are the functions implemented by the TMC and
        IFU boards:

          0 nothing (With the IFU this allows the prefetcher to start a
                request in this cycle)
          1 microdevice operation
            If Abus source is Lbus device, it's a read
                (I.E. if U AMWA 10 is deasserted)
            If Abus source is memory, it's a write with data from memory
                  (e.g. used when following pointers to load VMA from MD)
            Otherwise it's a write with data from data path
          2 start memory read
          3 start memory write
            (data must be on data path now unless it's a DMA write)
          4 TMC: increment VMA (don't use randomly, affects vma-offset
                logic)
            IFU: Reserve. This causes no memory control operation, but
                inhibits the prefetcher from requesting in this cycle.
          5 load VMA from Obus (via Lbus)
          6 block read (start read and on the TMC increment VMA. On IFU
                VMA increments only when the MD containing the data for
                this request is consumed by reading MEMORY-DATA-ADVANCE)
          7 block write (start write and increment VMA)

52-48           U SPEC <4:0>                Special Function
        This field enables a whole bunch of random kludges and features
        which didn't deserve their own microcode fields.

        In these descriptions, "#n" refers to U MAGIC <n>, bit n of
        the magic number field.

        0   R register gets Obus <4:0>.  #3 causes it to load from a function
            of the dispatch instead (this is for arrays).
        1   S register gets Obus <4:0>.
        2   Stack pointer gets Obus <27:0>.
        3   Frame pointer gets Obus <27:0>.
        4   Extra Base gets Obus <9:0>
        5   Data path control register gets Obus <7:0>:
                <1:0> = Stack Base = bits <11:10> of base/offset Amem address
                <2> = Sequence Break flag
                <3> = Trace Flag 1
                <4> = Trace Flag 2
                <7:5> not yet assigned
        6   Write special data path map memories:
                #0=0 => TYPE [U TYPE MAP SEL <5:0>, ABUS <33:28>] := BBUS <3:0>
                #1=0 => GC [ABUS <27:14>] := BBUS <3:0>
        7   [Rev-2 DP: Temporary instruction registers gets Obus <7:0>.]
            Rev-3 DP: Clear stack offset.
        10  Arithmetic Trap Enable
            #0 enables a trap if the Cond bit is set in the type map, and
            #1 enables a trap if the Bbus data type is not fixnum.
            #2 enables weird ALU functions (second set of 16)
            #3 (reserved for future use)
        11  Trap if Cond bit set in type map
        12  Trap if Cond bit set in type map or BBUS type not fixnum
        13  Multiply with type check (combination of 11, 12, and 17)
        14  Crocks decoded by the magic number field:
            1-7: GC Write Trap Enable
                #0  Slow jump to NAF if GC map says Abus points to any stack
                #1  Slow jump to NAF if GC map says Abus points to stack other
                    than the current stack.
                #2  (spare, needed anyway due to PAL limitations)
            10: Extended B-memory write address (allows writing all 256 locations)
            11-17: (reserved for future crocks)
        15  ALUB Sign Hack (if the shifter is used to LDB out the sign bit
                of the Ybus, it gets the complement of the sign bit instead).
        16  Crocks to Ybus.  U AMWA <11> selects between two words of
                miscellaneous fields as Ybus sources.  See the DPYSL2 print.
                It has to be possible to read the crocks without using the
                magic number, because you always extract a byte field, which
                requires the magic number and U AMWA <9:0>.
        17  Multiply
            #0  Multiplicand from Ybus <31:16>
            #1  Multiplier from Xbus <15:0>
            #2  Multiplier is signed (2's complement)
                Without #1, enables product onto Xbus.
            #3  Multiplicand is signed (2's complement)
        20  No special function.  This is the default value for this field.
        21  Addr from Abus  (physical address memory reference)
        22  Inhibit page tags (implies addr-from-abus)
        23  DMA (implies 21 and 22)
        24  Use PHTA  (map VMA through PHTC hash box instead of map cache)
        25  Check write access
        26  (not used)
        27  IFU control
            TMC5: #<1:0> =
                0: Load PC (must be given simultaneously with load VMA)
                        (This must be code 0, for (assign pc (word-pc ...)).)
                1: Load PC and force to odd halfword
                                (must be given simultaneously with load VMA)
                2: Start IFU (START-MEMORY of instruction-fetch)
                3: Increment PC (skip an instruction)
            IFU:  #<3:0>
                0: Load PC. (must be given simultaneoously with loading
                        the VMA, and resets the prefetcher and IFU)
                1: Load odd PC. Same, but force PC to be an Odd PC
                2: IFU Skip. Load IPC with incremented IPC. This should
                        only be used in hold mode
                3: IFU Skip Last. Same, but clear hold mode.
                4: Branch. Given on cycle where DP COND is the branch
                        condition.
                5: Write Decode LH. Given on on DATA cycle, writes IFU
                        "left half" decode memory from the memory data.
                6: Write Decode RH. Given on on DATA cycle, writes IFU
                        "right half" decode memory from the memory data.
                7: Reset MD Pipeline. Clears the MD loaded flags, and
                        flushes all requests in the pipeline.
                10: Restart. Given when doing START-READ for instruction
                        (Note: can't do NEXT-INSTRUCTION at the same time)
                11: Restart/Hold. Same, but enter hold mode
                12: Set Hold Mode
                13: Not used
                14: Advance. Advance to next byte in multi-byte instruction
                15: Advance Last. Same, but clear hold mode on last byte
                16: Accept PC. Load EPC from IPC (no other effect)
                17: Not used

        30  Arithmetic Trap Enable with Dispatch
            If a trap occurs, bits <11:10> of the trap address
            come from Abus <33:32> and bits <9:8> from Bbus <33:32>.
            Magic field same as function 10.
        31  Halt.  Machine stops after executing this instruction.
        32  NPC Magic.  Changes the meaning of U NPC SEL, also decodes
            magic number bits 1,0 as:
                0  not useful (drives garbage onto Lbus and into NPC in rev-2 SQ)
                1  NPC input to Lbus (use with microdevice read)
                   With rev-3 SQ, Lbus<19:16> get CSP, Lbus<23:20> get CUR TASK
                2  Lbus to NPC (use with microdevice write)
                3  nothing special (use to load NPC from CTOS)
        33  Awaken Task.  U MAGIC <1:0> select task 1, 2, 5, or 6.
        34  Write Task.  Obus<35:32> task number, <31:0> task state.
        35  Disable tasking.  Must be used twice in a row to work.
            (See the microcode manual for further discussion of the hair...)
        36-37  (reserved for the SQ board)

56-53           U MAGIC <3:0>               Magic number
        A 4-bit number used in many different ways.  Typically this
        is enabled by special functions.  The bits are described with
        the microcode fields that enable them.
```

```
         The bottom 3 bits select a dispatch, which may optionally be
         loaded into NPC and then branched to on the following cycle.
             0  ALUB <3:0> (bottom bits of shifter output)
             1  ABUS <35:34> (Cdr code of A bus)
             2  ABUS <31:28> (low type code, floating normalize, array registers)
             3  ABUS <25:22> (arrays)
             4  ABUS <21:18> (arrays)
             5  ABUS <2:0>   (8-way unrolled loops)
             6  BBUS <31:30>|ABUS<31:30>  (floating-point tags)
             7  (not yet assigned)

61-57            U COND SEL <4:0>         Condition Select
         Select condition for skipping or trapping
         Also used as byte size in some cases.
         0 :     CDR of the A bus is not = 0
         1 :     CDR of the A bus is not = 1
         2 :     CDR of the A bus is not = 2
         3 :     CDR of the A bus is not = 3
         4 :     From TYPE map
         5 :     B Bus type is not fixnum
         6 :     ALUB 0  (low bit out of shifter)
         7 :     YBUS 31 (Ybus < 0)
         10:     -GC TEMP
         11:     -GC THIS STACK
         12:     -GC OTHER STACK
         13:     ALU=0 (Bits 0-27 i.e. pointer field)
         14:     ALU not =0 (Bits 0-31 i.e. immediate number field)
         15:     ALU not =0 (Bits 0-33 i.e. all but cdr code)
         16:     not 28 bit carry
         17:     not 32 bit carry
         20:     ALU 31 (Obus < 0)
         21:     Sequence Break
         22:     Trace Flag 1
         23:     Trace Flag 2
         24:     -LBUS DEV COND (from selected micro device)
         25:     MC COND
         26:
         27:
         30:
         31:
         32:
         33:
         34:
         35:
         36:
         37:

63-62            U COND FUNC              Condition Function
         0  Ignore the condition.
         1  Skip if condition false.
            Bit 12 of the next microinstruction address comes from -COND.
         2  Trap if condition true.   Trap address is NAF.
         3  Trap if condition false.  Trap address is NAF.

67-64            U ALU <3:0>              Arithmetic/Logic Function
         Provides the mode, function, and carry for the ALU.  Also provides
         for the signed-compare and trap-if-overflow features.  These 4
         bits together with the special function "weird alu fcn" are
         looked up in a 32x8 prom.
         U ALU <3:0>      Normal func      Weird func
         0                Xbus             X+1-overflow (doesn't work)
         1                Alub             X-1-overflow (doesn't work)
         2                X+1              X+Y+overflow
         3                X-1              X-Y-overflow
         4                X+Y              X-Y-signed
         5                X-Y              X-Y-1-signed
         6                X+Y+1            NAND
         7                X-Y-1            ANDCY
         10               AND
         11               IOR
         12               XOR
         13
         14
         15
         16
         17
```

```
69-68         U BYTE F <1:0>              Byte Function
        Controls the shifter and masker, which provide the B input to ALU.
        This field is amplified by MAGIC, COND SEL, and AMWA as required.
        R is the number of bits of left rotation of the Ybus.
        S is one less than the number of bits selected by the mask, which
          come from the rotated Ybus.
        Merge means that bits not selected by the mask come from Xbus, otherwise
          they are 0.
        Rotate Mask means that the mask is rotated by the same amount as the
          Ybus (for dpb).  Otherwise the mask is right-aligned (for ldb).
        0  Shifter passes Ybus unchanged  (R=0, S=37)
        1  Weird kludges for multiplication and the PC, decoded from #:
           #2 = 0 => no rotate-mask, no merge, S=37, R from following table:
                0 -> 17         1 -> 16         2 -> 1          3 -> 0
                10 -> 37        11 -> 36        12 -> 21        13 -> 20
                Only codes 2, 3, 10, and 13 are used I believe
                     With code 13, S=17 and rotate-mask are specified.  These
                     are for the first cycle of a multiply.
                     Actually there is room for 8 random functions here.
           #2 = 1 => R=20,S=17  (i.e. operate on halfwords)
                     #3 => rotate-mask.  No merge.
                     #0,#1 may not be used (tied up by multiply)
        2  R=0, S=U COND SEL, no merge, rotate-mask don't care
           This extracts a right-adjusted byte from the Ybus.
        3  So-called general case.  Merge and rotate-mask from #2,3.
           #1,#0:
                0  R,S from U AMWA <9:0>
                1  R from RREG, S from COND SEL
                2  R from RREG, S from SREG
                3  R,S from INST (high 2 bits of S from COND SEL)
           SREG without RREG does not appear to be used

72-70         U OBUS CDR <2:0>            Obus CDR code select
        0  Abus <35:34>        (cdr code of Abus)
        1  Bbus <35:34>        (cdr code of Bbus)
        2  Bbus <7:6>          (for certain subprimitives)
        3  (illegal)
        4-7 Constant 0-3

75-73         U OBUS HTYPE <2:0>          Obus high type field select
        0  Abus <33:32>        (high type field of Abus)
        1  Bbus <33:32>        (high type field of Bbus)
        2  Bbus <5:4>          (for certain subprimitives)
        3  (illegal)
        4-7 Constant 0-3

76            U OBUS LTYPE SEL            Obus low type field select
        0  Magic number field  (i.e. 28-bit operation)
        1  Alu output  (i.e. 32-bit operation)

78-77         U CPC SEL <1:0>             Next microprogram address select
        0  NAF (microinstruction next-address field)
        1  CTOS (top of control stack, or IFU dispatch address)
        2  NPC (next microinstruction, or dispatch)
        3  (not used currently)

79            U NPC SEL                   Next next micro address select
        Normally:
          0  Dispatch (NAF except in bits 8-11)
          1  Next CPC+1  (next CPC controlled by U CPC SEL)
        With SPEC NPC MAGIC:
          0  CPC (address of current microinstruction, for traps mainly)
          1  CTOS (restore NPC from top of stack)
          (except that the NPC input can be forced from the Lbus instead)

93-80         U NAF <13:0>                Next Address Field
        Provides a microcode jump address, subroutine address, or
        trap handler address.

95-94         U SPEED <1:0>               Clock speed control
        180, 210, 225, 255 ns (I think)

101-96        U TYPE MAP SEL <5:0>        Type map select
        Selects one of 64 type maps to decode Abus <33:28>

109-102       U AU OP <7:0>               FPA control
```

```
        See the FPA if you want to know what these do.
        In a machine without an FPA, they are not connected to anything.

110             U SPARE                 spare

111             U PARITY BIT            Parity bit
```

Wiring for the second prototype backplane.

NOTE WELL: The silkscreen on the backplane is in error! The columns of 3 pins are named A,B,C from left to right; the silkscreen says C,B,A which is wrong.

As seen from the back, the left-hand 10-slot backplane contains the processor and the right-hand 10-slot backplane contains the bus. Consider slots 1-10 on the right-hand backplane to be numbered 11-20.

Slot assignments are as follows:

1	DP
3	SQ
5	MC
7	AU
9	FEP
11-20	Bus slots 0-9

Cards may not be plugged into the even-numbered slots on the left-hand backplane because of projecting wire-wrap pins.

Nomenclature: 3AC30 is slot 3 (third from the left), connector A (top connector), column C (right-hand column), pin 30 (third from the bottom).

All wiring is "straight across" unless specifically noted.

Ground is available on pins 3-20 of the center column of every connector. The rest of the pins in the center column are other power supplies, so be careful.

Etch cuts to remove unwanted SIP terminators:

(Make these cuts near both slot 10 and slot 20, where there is printed wiring to the two rows of holes, one on each side of the slot pins. Each signal is only wired to one or the other row of holes.)

BA29	LBUS FIRST HALF +
BC29	LBUS FIRST HALF -
BA32	LBUS CLOCK +
BC32	LBUS CLOCK -

Try to avoid covering this area with any wiring that is too tight to push out of the way, since SIPs will need to be soldered in (and may need to be changed as we experiment) and there may have to be one or two additional etch cuts in the vicinity of BC24.

[Still to do: SIP terminator stuffing instructions]

Bus wiring (between backplanes):

```
Wire 10AA01-30 to 11AA01-30
     10AC01-30 to 11AC01-30
     10BA01-28 to 11BA01-28
     10BA30    to 11BA30
     10BC01-28 to 11BC01-28
     10BC30    to 11BC30
     10BA29,10BC29 to 11BA29,11BC29   (use twisted pair)
     10BA32,10BC32 to 11BA32,11BC32   (use twisted pair)
```

All pins in columns A and C on the A,B connectors of slots 10,11 should be wired across.

Slot number isolation:

Cut etch on both sides of pins AA31-32, AC31-32 at every slot from 12 to 19. Half of these etches are on one side of the PC board and half on the other side.

Slot number wiring:

Wire 11AA30 to ground.

Wire AC32 to ground in slots 11, 13, 15, 17, 19.
Wire AC31 to ground in slots 11, 12, 15, 16, 19, 20.

```
   Wire AA32 to ground in slots 11-14, 19-20.
   Wire AA31 to ground in slots 11-18.

   Note: use separate wires at each slot, so that each slot can be individually
   rewired without affecting the other slots.

 Processor/bus isolation:

   Cut etch between 9AA30 and 10AA30                         FEP CONTINUITY / SLOT 4
   Cut etch between 8BA29 and 9BA29, 8BC29 and 9BC29         spare / LBUS FIRST HALF +/-
   Cut etch between 8BA30 and 9BA30, 8BC30 and 9BC30         TASK 8,9 REQ / CLOCK
   Cut etch between 8BA31 and 9BA31, 8BC31 and 9BC31         spare / CLOCK
   Cut etch between 9BA30 and 10BA30, 9BC30 and 10BC30       TASK 8,9 REQ / CLOCK
   Cut etch between 9BA32 and 10BA32, 9BC32 and 10BC32       LBUS CLOCK +/-

   Wire 8BA30 to 10BA30                                      TASK 8 REQ L
   Wire 8BC30 to 10BC30                                      TASK 9 REQ L
   Wire 9BA30,9BC30 to 10BA32,10BC32 with twisted pair       LBUS CLOCK +/-
   Wire 9BA31,9BC31 to 8BA32,8BC32 with twisted pair         PROC CLOCK +/-

 Processor internal isolation:

   Cut etch between 4AA13-28 and 5AA13-28                    LBUS ADDR <12:23>, etc.
   Cut etch between 4BC07-15 and 5BC07-15                    LBUS <36:43>, WITH ECC
   Cut etch between 4BC24-26 and 5BC24-26                    LBUS REQUEST etc.
   Cut etch between 4BC31 and 5BC31                          COND L

 Processor Internal Wiring:

         Every pin in the A and C columns of the C and D connectors of slots
         1, 3, 5, and 7 should end up with exactly one wire on it, with the
         following exceptions.
         No wires:  7DC01-32
         Two wires: 3CA01-5      5CA01-5       7CA01-11
                    3CA12-32     5CA12-32
                    3CC01-15     5CC01-15      7CC01-11
                    3CC29-32     5CC29-32
         There should be exactly one wire on 9CA1-11 and 9CC1-11
         and no wires on the rest of the C and D connectors of slot 9.

 The following pins are to be wired across on slots 1, 3, 5, 7, and 9.
 Use twisted pair on these two pairs:

         CA01 & CC01                                         PROC WP +/-
         CA02 & CC02                                         PROC FIRST HALF +/-

 Wire these with ordinary wire:

         CA03-5                                              Clock control, next inst
         CC03-11                                             misc global signals

 The following pins are to be wired across on slots 5, 7, and 9:

         CA06-11                                             Bus control

 The following pins are to be wired between slots 1 and 3:

         CA06-11                                             U AMRA 0-5
         DA01-32                                             Microcode

 The following pins are to be wired across on slots 1, 3, 5, and 7:

         CA12-32                                             Microcode
         CC12-15                                             spares
         CC29-32                                             Abus 32-35

 The following pins are to be wired between slots 1 and 5:

         CC16-28                                             IFU, bus control

 The following pins are to be wired between slots 3 and 7:

         CC16-28                                             AU control

 The following pins are to be wired between slots 3 and 5:
        DC01-32                                                  SQ/MC communication

Wire the following:

        1DC01-32 to 5DA01-32                                     Abus
                (NOTE: That's 1DC to 5DA)
```

Instructions for setting up L-machine "June" backplane -- Moon 10/1/82

- Power wiring

Check that all power connections are tight. They should not be tightened as if you were putting a wheel on a car; they should just not be loose.

-2V power is missing to the two ECL SIPs in the B row:

At B on the left, solder a wire from -2V to the trace coming out of the bottom SIP position.

At B on the right, solder a wire from the top of the top capacitor to the top pin of the SIP you are about to insert into the bottom SIP position. (It seems to be too hard to solder to the socket without filling the hole with solder.)

- Power verification

Power voltages are on the middle of the 3 columns of pins at each connector. The top 2 and bottom 2 pins have the voltage that comes in on a screw connector at the left. The remaining pins are ground above and +5 below, as indicated on the silkscreen. Don't worry about the arith slot for now. There are no errors in the silkscreen.

- ECL terminators

Install 8-pin 68-ohm ECL SIPs in the bottom position at each end of row B, and in the only position at the left end of row C. The end with the bulge (the bypass capacitor to ground) goes down. I believe the dot is usually on this end, but sometimes it is on the other end. Install a total of 3 SIPs.

The following pins should be at -2V relative to ground, if the SIPs are installed right-side-up:

```
        1 BA32,BC32,CA1-3,CC1-2
        14 BA29,32,BC29,32
```

- TTL terminators

Install 8 220/330 10-pin SIPs at the right-hand end of the A row. To verify that they are in right-side up, power on the machine and check that these pins are at 3 volts relative to ground. If they are at 2 volts, the SIP is upside-down. Dots on the SIPs are not reliable.

On the left-hand end of the B row, install in the second-to-the-bottom slot a 10-pin xxx SIP. xxx is nominally 270/560, however 220/330 will do as will 330 or 470 (pullup only rather than voltage divider). Pin 1 goes up if the SIP is a plain pullup.
This SIP should bring pin BC20 to between 3 and 5 volts with respect to ground (3 volts for 220/330, 3.4 volts for 270/560, 5 volts for pullup).

On the right-hand end of the B row, the SIPs are arranged like this:

```
                 ||| 2 3
               1 |||   4
                 ||| 5
                 |||
               6 ||| 7
```

Install SIPs as follows:

1. 220/330. Should bring pin BC10 to 3 volts. (2 volts means it's upside-down).
2. 220/330. Should bring pin BC1 to 3 volts.
3. 270/560, however 220/330 will do since currently we are not terminating the other end of the spy bus. Should bring pin BA1 to 3 volts.
4. Same as 3, should bring pin BA10 to 3 volts.
5. Leave empty (the task requests are terminated only at the sequencer end).
6. 220/330. Cut off and tape over pin 3 (pin 1 is at the bottom) to avoid terminating TASK 9 REQ L (BC30) twice. Should bring pin BC24 to 3 volts.
7. The 68-ohm ECL SIP already installed.

There should not be any SIPs plugged in elsewhere.

- Temporary terminators

```
Install a 330-ohm resistor between BA18 on the sequencer slot, and +5V
(middle pins near the bottom of that connector).  This will go away as
soon as the appropriate ECO to the SQ board (#6) is made and installed.

- Slot-number wiring

This goes at the bottom of the A row on all nine Lbus slots.

Wire ground to the following pins:

7 AA30                  puts all 9 slots in the bottom group of 16.
14 AA31,32,AC31,32      slot 0
13 AA31,32,AC31         slot 1
12 AA31,32,AC32         slot 2
11 AA31,32              slot 3
10 AA31,AC31,32         slot 4
9  AA31,AC31            slot 5
8  AA31,AC32            slot 6
7  AA31                 slot 7
6  AA32,AC31,32         slot 8

-------
```

As a standard feature, the 3600 console is provided with a digital audio output system. The console presently offers only monaural conversion but will be augmented by a full 16 bit/channel stereo converter at some future date. (For a price, of course.)

Audio output is a function of a microtask, driven by a microtask wakeup signal sent from the I/O board (IOB). Stereo audio samples are 32 bits in length, the most significant 16 bits being the left channel, the least significant 16 the right. With 16 bit stereo samples, approxomately 5% of the processors time is used in outputting pre-computed audio samples at a 50 khz sample rate.

The digital audio output from the processor is double-buffered on the IOB. Since the audio shift register takes 20 microseconds to shift out a full stereo sample to the manchester encoding logic, the processor has this much time to foreward the next sample to the buffer register.

The serial audio data from the shift register has included in it additional bits which facilitate easy decoding at the console end. The bit stream is as follows:

```
1  start bit
16 bits of sample
1  channel bit (L)
2  stop bits

1  start bit
16 bit of sample
1  channel bit (R)
2  stop bits
------------------
40 total for a stereo sample
```

All 40 of these bits must be transmitted in one sample time. For 50 khz sampling frequency, this gives a 2 mhz bit rate. The bit clock comes from a plug-in crystal oscillator on the I/O paddle card. The frequency of this oscillator is 80 times the desired audio sampling frequency (twice the bit rate). If connection to external digital audio equipment is desired, the oscillator can be left out and the board will accept an external clock via a card edge bnc connector.

Manchester encoding allows audio data to be transmited along with the bit-clock down a single serial line to the console. At the console end, the serial data stream is decoded and the clock is extracted. The least significant 12 bits of a stereo sample (the right channel) are sent to a 12 bit digital-to-analog converter (DAC). The analog output from the DAC is amplified and sent to a small speaker in the console. A connector is also provided to tap the converted analog audio signal.

For a detailed examination of the operation of the audio circuitry please refer to <LMIOB>ABUFR.DRW, ATASK.DRW, ATASK1-REV4.PAL, AND ATASK2-REV4.PAL for the drawings and the control PAL files respecively.

Layout services

Triad Engineering Corp, Burlington Mass 273-1880 -- PC artwork design
Approx $10K per board for layout.

Algorex Corp., Syosset NY. PCB layout service. Specializes in complex boards, high speed ECL. Automatic layout and routing, followed by interactive circuit simulator that checks for transmission line effects of stubbing, net impedences. Testability analysis.

Automated Images, Inc. Woburn Mass 933-1731. Joanne Litchfield.
PCB design and tooling. Photoplotting, Gerber 6240 Photoplotter.

Fab houses

Altron, Lowell, MA

CircuitWise. North Haven Conn. Rollin Mettler

ÉCC, Holden, Mass

ERC, Waltham MA (?) Small outfit

Hadco, Nashua, NH, Derry NH Hod Irvine

MBC Engineering.
Quote on 5 layer board, small # of hole sizes, clearance 40 mil hole, 60 mil clearance. 25 pcs, 7 weeks, production 16 weeks, tooling $400, solder mask $35/side. $85-95 each in 100's.

Metropolitan Circuits, Bridgeport NJ Larry Velie

Multaplex, Santa Clara (408) 727-4033, reped by Vector Sales, Ben Rizza, 787-2790.

Photocircuits

Printed Circuit Corp, Waltham Mass. Pete Sarmanian?

Qtronics, Nashua, NH

Rockwell International/Collins. Cedar Rapids, Maine(?) Meridith Suhr

Wellborn Industries. (800)-247-5972. Single and double sided boards. Going to multilayer. 8,10 mil line widths. Stan Gentry, President. Fast prototyping service, 2-3 weeks from artwork. Special premium gives 1 week turnaround. Stan Nolan is rep, A-FAB sales, Rochester NY (716) 244-8248. In business 2 years, now at $3 million.

```
Console cable protocol
 - 7 differential pairs (TTL) as follows:
 - Data (4 bits at 12 MHz for black&white, 40 MHz for high-resolution color)
 - Clock (6/20 MHz square wave, both edges active)
 - Tag (8 bits for every 32-bit data frame, see below)
 - Input (standard UART, 100K-1M bit rate)
   (Must be slow enough so FEP does not lose incoming characters)

Console hardware
 - phase-locked-loop at 8 times frequency of clock on cable
   controls 4-bit video shift register
 - rising/falling edge compensation for video (no! must be prior to
    inverse video)
 - tag decoding
 - generation of horizontal sync and blanking
 - keyboard scanning, mouse tracking (send deltas periodically)
   provide for attachment of other input devices to console microprocessor
 - audio output
 - maybe an LED that lights if valid clock, tags are being received
   for maintenance (checking whether cables are broken when an
   installed terminal stops working)

Tag codes
  11v0h0x0 are the 8 tag bits for a frame.
  The 11 synchronizes the frame boundaries.
  v is the vertical sync signal; it goes direct to the monitor.
  h is the horizontal sync signal.  A 1 here means that this frame is
    the last video frame in the current raster line, and the next frame
    is the audio data.  The horizontal sync to the monitor needs to
    be derived from this with some logic.
    Consoles that need hairy sync patterns (equalizing pulses and so on)
    have to generate them in the console, probably.
  x is available for future use.  Could be serial data to console micro?
  (Note that the horizontal line rate needs to be an integral multiple
   of a frame.)

Patch panel
 - small installations want to be able to have an inexpensive manually
   operated patch panel between terminals and machines
 - suitable connectors (keyed, mechanically strong, place for labelling)
 - LED powered by extra wire from computer, lit if someone is logged in
 - ((compare the MIT 8th floor patch panel!))

Video Switch
 - modular and expandable
 - controlled by what?  (Stripped L machine or 68000)
 - on network?  (provides a way to send commands, may be unnecessary)
   could also send commands over a computer-to-console UART channel
 - local video buffers (for talking to timesharing machines)
 - Should there be a cheap switch that just switches, and an expensive
   one that has local video buffers, network, EIA lines?
 - switch attention (user typing commands at switch)
 - informing machine of console type and location
   special characters sent down keyboard input line from video switch
 - high-resolution color uses two channels, (low-res color uses one)

Color
 - color map and DACs are in console (no long-distance analog transmission)
 - color map stored in video buffer after picture, whole map sent to
   console during every vertical retrace (1536 8-bit pixels)
 - console can perhaps include a general interface for random devices
   to pick up other data during this time

Audio
 - two 16-bit words (stereo) per video raster line
   default console with crummy speaker only looks at 8 bits from left channel
 - buffer in horizontal-retrace section of raster lines of video buffer
   microcode task fills this buffer ahead of video refresh
   Thus the standard TV hardware in the machine doesn't really know about the
   audio, it just transmits the whole video buffer, some of which is video,
   some of which is audio, some is perhaps other stuff

Video Input
 - same protocol, but camera is source and computer is sink.  Can go
   through video switch (which can know which cameras are near which
   terminals and switch both to same computer).
```

This file specifies the bus as seen by the memory card.

DATA<47:0> - bidirectional data lines. Only the low 43 bits are used in a 36-bit machine with 7 ECC bits.

ADDR<23:0> - physical address from processor.

SLOT NUMBER<4:0> - built into the backplane. See discussion below.

MEM TO BUS - from processor, enables selected memory card to drive DATA lines. The source of data is a register which is clocked as CAS to the RAMs ends. The card which responds to MEM TO BUS is the one which was most recently selected, not the one currently selected; ADDR and MEM RAS may change while MEM TO BUS is asserted.

DISABLE ECC - from memory, driven same time as DATA and specifies that no ECC is present. For the TV.

FAST CLOCK - from processor. This is the basic clock which is divided down to yield all other clocks in the machine, other than the inherently asynchronous ones in the I/O devices. This clock runs at a 20-25 MHz rate probably. (40-50 ns).

MEM RAS - from processor. See below for discussion of how this is used.

MEM CAS - from processor.

MEM WRITE - from processor.

MEM REFRESH - from processor, causes all memories to respond to MEM RAS and causes each memory card to RAS all of its chips, not just one row.

MEM WAIT - from memory when it is busy. The TV uses this to make the processor wait. The normal memory card does not use it. The processor registers MEM WAIT at the end of the FAST CLOCK cycle after MEM CAS is asserted, and inserts additional FAST CLOCK cycles until MEM WAIT goes away, holding all bus signals other than the clock constant in the meantime. This makes the minimum width of MEM CAS 3 fast clocks.

MICROINSTRUCTION CLOCK - If this is asserted, the next FAST CLOCK represents the boundary between two microinstruction cycles. The memory card does not look at this, but DMA I/O devices do.

The same bus is used to talk to I/O devices as well as to memory. When talking to an I/O device, the MEM RAS and MEM CAS signals are not used; a different set of signals is used. There may possibly also be some I/O devices, in addition to the TV, which respond to the bus as if they were memory. They will respond to ADDR<23:19>=37 and will use the lower address bits to select the device. Should they also use slot numbers, so that there can be more than one device of a given type, and if so exactly how does the software automatically discover the hardware configuration?

This is based on the theory that all the high-level timing is decided in the processor, which knows whether cycles are being stretched, when page-mode is to be used, when an extra RAS is necessary because a sequence of page-mode cycles has been interrupted, when it is safe to refresh, etc.

This timing is in multiples of FAST CLOCK. The memory card registers the MEM RAS, MEM CAS, MEM WRITE, MEM REFRESH, and BOARD SELECT (decode of ADDR, MEM RAS, and SLOT NUMBER) signals on the active edge of FAST CLOCK. It then generates the internal RAS, CAS, and WE signals to the RAMs based on the bits in these registers and internal delays designed to provide the proper timing for the RAMs and to allow enough setup time for the address and data. ADDR is stable on the bus by (a certain time before) the time MEM RAS is clocked into the register, and remains stable during the full cycle, except that the low 8 bits can increment at a defined time. The write data is stable on the bus by the time MEM CAS is clocked into the register. The memory card takes care of delaying RAS and CAS long enough to set up the address and data at the chips. The register for the control signals on the memory card eliminates bus skew from this calculation. Each memory card knows whether it is selected; this is a 1-bit register clocked by FAST CLOCK which changes only when MEM RAS is asserted and which loads from a comparison of ADDR and SLOT NUMBER. Thus a memory card remains selected after a cycle completes until the next cycle starts; this is necessary for overlapping of MEM TO BUS.

The TV uses the same interface in a different way. MEM CAS is a cycle request, and a full cycle is always performed (no page-mode), requiring the use of MEM WAIT. MEM RAS serves as a warning that one or more cycles will be requested very soon. Thus MEM WAIT serves both to make the processor wait when the TV memory is busy generating video, and to make the processor wait for the slower cycles of the TV memory.

Timing for a simple read cycle (T0 = first half of microinstruction cycle, T1 = second half of microinstruction cycle). Here we assume that FAST CLOCK is 50 ns and a microinstruction cycle is 200 ns. These assumptions are arbitrary and guaranteed to change. The FAST CLOCK period will probably not be a nice round number, but will be chosen to divide various required times nicely. A time "m.n" means microinstruction cycle m, FAST CLOCK cycle n. T0 consists of x.0 and x.1; T1 consists of x.2 and x.3.

```
1.0-1.1 Virtual address being mapped, or physical address coming from Amem.
1.2     ADDR being driven on bus.  MEM RAS being driven on bus.
1.3     Memory card registers control signals, and after a short
        delay sends RAS to the chips.  Memory cycle starts roughly
        25 ns before the end of microinstruction cycle 1.
        [This may want to be changed so the ADDR and MEM RAS occur
        later, allowing more time for the virtual address map, if the
        processor cycle time is long enough so that there is enough
        time for a memory cycle.]

2.0-2.2 MEM CAS driven onto bus.  Thus CAS goes to the chips in
        2.1-2.3.

3.0     MEM TO BUS asserted, DATA lines being driven with read data.
        MEM RAS being driven if cycle is going to continue with an
        additional CAS, otherwise MEM RAS goes away during this time.
3.1     Memory data starts into datapath towards the end of this time.
3.2-3.3 Processor drives memory data back onto DATA lines, for benefit
        of DMA I/O devices.  In the event of an ECC error, the corrected
        data is driven, and extra FAST CLOCKS may be taken.
```

Note that another memory cycle can start during 3, thus ADDR can change at 3.2 and MEM RAS can be asserted again on the bus starting in 3.2.

For a block read, another CAS has to occur during cycle 3. The MEM CAS line on the bus is deasserted during 2.3 and 3.0, then asserted in 3.1 through 3.3, and deasserted in 3.4 (corresponding to 2.3). 3.4 is an extra clock inserted by the processor to provide enough time for a page-mode cycle in the RAMs. ADDR increments at the start of 3.1 and is stable by 3.2 when CAS happens. MEM RAS is continuously asserted through the whole block read, shutting off during x.0 of the last microinstruction cycle.

Timing/duty cycle of registered bus signals. The signals seen by the RAMs are altered somewhat from this. In particular the onset of RAS needs to be delayed by about 25 ns.

```
MEM RAS asserted time = 1.3 through 3.0 = 6 clocks = 300 ns
MEM RAS precharge time = 3.1 through 3.2 = 2 clocks = 100 ns  (too short)
MEM CAS asserted time = 2.1 through 2.3 = 3 clocks = 150 ns
MEM CAS precharge time = 3.0 through 4.0 = 5 clocks = 250 ns  (non block mode)
MEM CAS precharge time = 3.0 through 3.1 = 2 clocks = 100 ns  (block mode)
```

For a write, the write data, with valid ECC bits, must be at the RAMs the same time as CAS. Thus it is driven on the bus during 2.0 through 2.3--this gives very little time for computing the ECC bits; they are actually computed during 1.3 and registered at the same time as the data. ECC computation overlaps with ALU=0 and other flags computation. The memory card probably does not register or latch the write data; it only buffers it.

When doing a read-pause-write, the write data comes out of the datapath during 3.3, goes on the bus during 4.0 and another CAS occurs during cycle 4. This is the most common form of write, used for Lisp SETQ, ASET, RPLACA.

Discussion of card addressing and slot numbers.

The idea is that there are no address dip switches anywhere; instead the address of any card depends on what backplane slot it is plugged into. Furthermore you can have multiple copies of a card in a system and they will automatically be at different addresses. This applies both to memory cards and to I/O cards; it does not apply to processor cards, which probably have their own dedicated backplane slots.

The bus is used in three ways; accessing memory, accessing I/O device registers which look like memory, and accessing "special devices". Special devices are distinguished because they do not look like memory and are accessed by special microcode. One example of a special device is a floating-point accelerator. Another example is a DMA device such as the disk; the DMA microcode task commands the disk to put data onto the bus or take it off, while doing a memory cycle. I/O device registers look exactly like memory, a la Unibus or Xbus. A portion of virtual memory is mapped into the physical addresses for I/O devices and accessed through normal subprimitives. An I/O device ignores MEM RAS and responds to MEM CAS as if it were MSYN. Note the need to have an output register and respond to MEM TO BUS.

Each backplane slot has 5 pins which are its slot number; they are grounded in a different pattern at each slot, and pulled-up on the card. Slot number 37 cannot exist because that number is used to mean special things.

Each memory card, and other memory-like device such as a TV, responds to the physical address which corresponds to its slot number in bits 23:19 (for 512K memories).

Each I/O card responds to the physical address which has 37 in bits 23:19 and its slot number in bits 18:14. This gives each I/O card 16K of address space for its device register. Device register 37777 on every card is a device ID register, which can be read to determine what kind of board is present in that slot and what options it has. When the machine is booted, the configuration of memory and I/O cards is automatically discovered and tables of the addresses of the important devices are set up. Either this is done by the diagnostic microprocessor before trying to boot the Lisp engine, or the basic devices required to run Lisp code (some memory and the boot disk) have to be in fixed slots, and the remaining configuration is determined by Lisp code. It may be a requirement that slot 0 always contain a memory card if there are any wired-in physical memory addresses. Memory cards can be depopulated, so the size of each memory card is also automatically discovered during booting.

Probably memory cards do not bother to respond to their I/O address space; the device ID for a memory card and for an unoccupied slot are the same (undriven bus) and one distinguishes them by testing response to the memory address space for that slot.

TV cards respond to both their slot's memory address space and their slot's I/O address space. The video buffer is accessible as memory, and any control registers are accessible in the I/O space.

The I/O address space for slot 37 is used by the processor. The internal A memory appears here (so that virtual memory can be mapped into it) along with some devices, such as a clock, that are built into the processor. Possibly there may also be some magic control registers for use by the diagnostic microprocessor.

Remain to be specified:

Use tristate or open collector?
Signals asserted low or high?
Standardize on which parts as drivers and receivers?
What termination values on which signals?
Connector pin assignments (78 signals)
How much bus skew should be allowed for?
Will refreshing all memory cards simultaneously cause too much of a power-supply transient? This seems to depend on the brand of RAM; some use about the same amount of power refreshing as in a normal cycle and some use less than 1/10 the power. In the normal case we are talking about 43x8x0.050=17 amps per memory card.
Should the clock be distributed in ECL? Should there be some provision to adjust for clock skew? Should the bus clock be deliberately

delayed relative to the processor clock so that we always know in
which direction the skew goes?

The reaction of +5 at refresh time seems to indicate not enough hi-freq filtering.

Look for ground noise on the array, and at the driver chips, and the bus driver chips.

Try additional .1uF and 1uF soldered in the array to see how much the refresh VCC bump is reduced. (Short leads please!) See if the bump exists at the caps themselves, and how fast it increases as you move away from the caps (to judge the "effective radius" of the bypass cap). This should eventually be under .2V. (The chips have a +/-10% spec on VDD, but they are often more sensitive to a glitch than an out of tolerance voltage.)

[See scope photos, which show -.4V spikes during refresh cycles, measured across the RAM. Noise is much the same over the whole board. We will add holes for bypass caps above each RAM in the next revision of the artwork. Someone should investigate which cap to use -- a mixture of caps might be more optimal than just one.]

The data output registers, the F374, have longer closely packed runs. Is there any evidence of adverse coupling there?

[Still need to look at this]

Some things I think should be looked at:

```
  The delay from LBUS CLOCK to RAS and CAS at the chip.         [19 ns]
  The delay from LBUS address to the chip.
  The delay from CAS MPX to the address at the chip.            [14 ns]
  The delay from LBUS CLOCK to data on the bus.
```

Also the risetime of the signals in the chip array: address, RAS, CAS, WE, DO. The waveform at the near and far end of the etch traces.

[Look pretty good, with the following exceptions: The address muxes (74F258) fan out to 22 loads of Am2966. There is a 1V "step" on the rising edge of the mux output. The fanout should have been restricted to 11 loads, but we may be able to live with things as they are.]

Inserting a series R in the RAS/CAS lines to reduce the undershoot. In the 10-20 ohm range I suspect, find the minimum resistance that will get the overshoot down to .5V, and measure the increase in rise/fall time.

Try parallel termination of the RAS/CAS/WE lines. [100 ohms to ground seems to work pretty well, see scope photos.]

[We will modify the layout to allow for series terminations on the RAS/CAS/WE lines. We will also put in artwork for SIP parallel termination of those lines at the back of the board. We'll choose which one to use by further experiment.]

Look at data in (DIN) lines, they are LS240's, and drive 8 loads scattered over the board.

Change logic of ROW FF to obey the "correct" timing spec. Choose the right delay taps.

Change the logic of Enable Out so that it is generated faster. Look at ENABLE OUT <43:8> L for signal problems.

```
Approximate breakdown of control-memory usage (tmc5-mic 225):

  (this is out of a grand total of 6651.  There are also 719 halt
   instructions planted at the dispatch addresses for undefined opcodes.)

Arrays                             654    (< 100 for two-dimensional arrays)

Function call instructions         431
Function call support              126
Argument-taking instructions       316
Function return instructions        79
Stack-manipulating stuff            83    Total 1035

Message pass (instance call)        76
Flavor instructions                 49
Special variable binding            70
Catch/throw                         65    Total 260

"Control"                          172
Traps                              129
Stack group switch                 116
Stack buffer control                81    Total 498

Bitblt                            1023
Float (single and double)         1729
Disk                               340
Network                            172    Total 3264

Basic instructions more suitable for hardwiring:

Arithmetic                         243    (includes some escape-to-macrocode stuff)
Multiply/Divide                    168
Data Movement,Predicates           148    (includes car/cdr/rplaca/rplacd)
Subprimitives                      116
Branches                            53
Consing                             19
GC primitives                       89
Map, phtc, page tag primitives      72
Virtual address map cache miss      34    Total 942

Grand total 6653 (I'm off by 2 somewhere)


Approximate breakdown of usage of processor RAMs:   (widths include parity)

Control memory                   8K x 112           (option to expand to 16K)

Task state                       16 x 36

Microcode stack                  255 x 16       (1K RAMs, 255 locations usable)

A-memory                         4K x 40
   Stack buffers                 2048
   Micro constants                 60           (60 used, 256 allocated)
   Micro variables                 72           (72 used, 256 allocated)
   System communication areas     117           (117 used, 512 allocated)
   Not allocated                 1024

   I believe some microcode variables exist that aren't listed here
   because they don't have names in the micro compiler (various history buffers).

B-memory                         248 x 40       (1K RAMs, 248 locations addressable)
   Micro constants               102
   Micro variables                43

Type Map                         4K x 4         (3520 used currently)

GC Map                           16K x 4        (expands with virtual memory)

Page tags                        32K x 3        (expands with physical memory,
                                                 should be 64K for 24-bit physical addr)

Virtual address map cache        8K x 36

Instruction cache                ?
```

Lbus and Backplane Documentation -*- Text -*-

Table of Contents

Nomenclature and General backplane conventions

NOMENCLATURE

slot connector row pin

Slot is 1-10 (or however many card slots there are in the backplane)
Slot 1 is on the left when looking at the backplane from the rear;
it's on the right when inserting cards.

Connector is A-D. A is at the top.

Row is A-C. Row A is on the left when looking at the backplane from the rear.

Pin is 1-32. Pin 1 is at the top.

POWER

All power connections are on the "B" (center) column of the DIN connectors. Most of the pins are used for GND and +5. On each connector, a pair of pins at either end are used for connecting other voltages.

Pin	Conn A	Conn B	Conn C	Conn D
1-2	-2.0	-2.0	-2.0	-12
3-20	GND	GND	GND	GND
21-30	+5	+5	+5	+5
31-32	-5.2	-5.2	-5.2	+12

-2.0V is mostly for the AU and the clock logic on the FEP board. It is also used to terminate the ECL clocks on the backplane. The -5.2 is on pins 31-32 so that the AU slot can easily convert pins 21-30 to -5.2. The AU card and slot will have different connector spacings so there will be no possibility of plugging a board into a slot that has the wrong voltage on 21-30. [Maybe some form of mechanical keying also?]

[These voltage bus assignments differ from the pre-prototype which had -5.2 on BB31-32 and DB31-32, and -2.0 on CB31-32]

The B row of the DIN connectors should have insulated or very short pins so that the risk of shorts from power to other power or signals is minimized.

I/O CONNECTORS

Connectors C and D on non-processor slots are non-bussed and used to connect to paddle-boards.

CARD CAGE EXTENSION

[Formerly there was a scheme by which AA1-30, AC1-30, BA1-30, and BC1-30 could be extended to a second card cage, using 8 60-conductor flat cables with alternate grounds. This has been flushed as electrically undesirable and unnecessary given the full-width card cage with 20 or more slots.]

```
Pinout for Lbus slots

All signals are bussed across through all Lbus (memory/I/O) slots unless
otherwise noted.  All signals are terminated at the end of the bus, however
the termination values that haven't been figured out yet are not listed.
Signals are listed in pin order.

LBUS ADDR 0-23           AA1-24          Backplane 220/330 ohm
        Address to be referenced; valid late in the cycle (prior to
        high phase of clock).  Bits 1-0 select a bank on the memory
        card; bits 9-2 a column; 17-10 a row; 18 a group of 4
        interleaved banks; 23-19 a card (see slot discussion below).
        These must be driven with 64 ma drivers.
        LBUS REQUEST L enables all of these bits.
        LBUS REFRESH L enables bits 17-10.
        LBUS ID REQUEST L enables bits 23-19, 6-2.

LBUS ID REQUEST L        AA25            Backplane 220/330 ohm
        Enables selected card to drive the byte of its ID prom selected by
        LBUS ADDR 6-2 onto LBUS <7-0>.

LBUS BLOCK REQUEST L     AA26            Backplane 220/330 ohm
        Notifies the memory that this request cycle is to a sequential
        address.  Only meaningful if LBUS REQUEST is asserted.
        This pin used to be a spare.

LBUS DEV READ L          AA27            Backplane 220/330 ohm
        Enables the card selected by LBUS DEV 9-5 to drive data selected by
        LBUS DEV 4-0 onto LBUS 43-0 (usually 31-0, with dtp-fixnum supplied
        in bits 35-32).

LBUS DEV WRITE L         AA28            Backplane 220/330 ohm
        Enables the card selected by LBUS DEV 9-5 to execute the command
        selected by LBUS DEV 4-0 and to receive Lbus data, at the clock edge.

LBUS DEV COND L          AA29            Backplane and processor 220/330 ohm
        Microcode-testable condition drivable by device selected by LBUS DEV
        READ or LBUS DEV WRITE.  Driven by a 48ma open collector driver.

LBUS SLOT 4-0            AA30-32, AC31-32
        5-bit unique number in each Lbus backplane slot.  These lines are not
        bussed across.  At each slot they are grounded or open to provide the
        appropriate number; the card pulls them up and matches them against
        LBUS ADDR 23-19 and LBUS DEV 9-5.
        [Perhaps LBUS SLOT 4 is bussed across and left open; a ground wire
        is then added in the main backplane and omitted in the expansion
        backplane.  This depends on whether we ever do an expansion backplane.
        Probably this idea should be flushed.]

LBUS 0-43                AC1-30,BC1-14   Backplane and processor 220/330 ohm
        Tristate data lines, for both memory and microdevices.
        LBUS 43 is a spare ECC bit not used by the current processor.
        These must be driven with 48 ma drivers.

        Note that these lines must be driven with data that is synchronized
        to the Lbus clock.  The IO board violates this in numerous places,
        but it is wrong and does not work.  The Lbus data lines must be stable
        from partway into First Half through the end of First Half, or
        synchronizer failure in the latch in the memory control will feed
        garbage into the data path, causing an eventual parity error.

        [This can't be doubly terminated because there must be drive margin for
        20 or so PNP card loads -- about 10-12 ma.  Maybe the processor has
        slightly weaker terminations??]

SPY 0-7                  BA1-8
        Tristate spy and FEP-DMA bus.  Used for diagnostic purposes.

SPY ADDR 0-5             BA9-14
        Address for spy bus.

SPY DMA SYNC↑            BA9
        Driven by device; rising edge clocks data from SPY 0-7 into device or
        from SPY 0-7 into FEP's buffer, and advances the buffer pointer.  The
        device drives this with an open-collector driver (only for
        convenience).  This is the same signal as SPY ADDR 0.

SPY DMA BUSY L          BA10
        An open-collector signal, terminated at the processor and driven
        by either the FEP or the FEP-DMA device.  When this line is deasserted
        it signals the end of the transfer.
        This is the same signal as SPY ADDR 0.

SPY READ L              BA15
        Enables diagnostic data selected by SPY ADDR 5-0 onto SPY 7-0.

SPY WRITE L             BA16
        The trailing (rising) edge clocks diagnostic data from SPY 7-0 into
        the diagnostic register selected by SPY ADDR 5-0.

SPY DMA ENB L           BA17
        Asserted low to enable use of the spy bus for FEP-DMA.

TASK 4 REQ L            BA18
        Low-priority task request (used by multiple devices).

L̂BUS DEV 0-9            BA19-28         Backplane 220/330 ohm
        Device number for microdevice operations.  Bits 9-5 select a card,
        bits 4-0 select a subdevice, and LBUS DEV READ L and LBUS DEV WRITE L
        are the enabling signals.

LBUS FIRST HALF +,-     BA29,BC29       68 ohms to -2 V
        Differential ECL signal which selects between the two phases of a
        microinstruction cycle.  This is allowed to be skewed so that it is
        slower than the clock, and should not be used as an edge-triggered
        clock.

EXTERNAL REQUEST L      BA31            220/330 on MC board
EXTERNAL GRANT L        BC31            220/330 on MC board
        External request is asserted by an LBUS device (like the DMA
        adaptor) synchronous with lbus clock.  This signal is clocked
        on the MC board and a priority decision is made between FEP,
        refresh, external, processor, and IFU requests.  The MC
        asserts EXTERNAL GRANT L at the beginning of the cycle
        available to the external device.  The device should use this
        signal to enable it xcvrs onto the bus.  If more than one
        device need to use this mechanism, they will have to arbitrate
        between themselves.
        This signal has to be jumpered around the FEP (and AU) card
        slots.  Also, the trace between SQ and MC will have to be cut.

LBUS CLOCK +,-          BA32,BC32       68 ohms to -2 V
        Differential ECL clock.  The trailing (+ falling) edge, after passing
        through a 10125 and a 74F-series invertor or gate, is the clock for
        all edge-triggered registers.  The leading (+ rising) edge starts RAS
        in the memory card.  A card may only supply 2 units of 10125 load
        to LBUS CLOCK and LBUS FIRST HALF.  [This may be modified on some
        boards where the clock fanout tree needs to be larger.  I worry
        about multiple Lbus loads slowing down clock, or introducing skew
        between clock and first-half.]

LBUS WITH ECC           BC15            Backplane and processor 220/330 ohms
        Pulled down by a memory-mapped device when it is driving LBUS 35-0 but
        not LBUS 36-43.  Pulled up the rest of the time by a termination at the
        end of the bus and on the MC/IFU card.  Driven with a 48ma driver.
        Note that there is always a 1-cycle delay between accesses to different
        types of memory, due to interleaving rules, and therefore the rise time
        of this signal is not critical.  The fall time, however, is critical.

        This signal tells the memory control that an ECC code is present and
        should be checked, if it is high; or that no ECC code is present and no
        error detection or correction is to be performed, if it is low.  In the
        latter case, the LBUS data normally comes from a device rather than a
        memory and only LBUS 35-0 are valid; often LBUS 35-32 are 0001 (fixnum
        data type) and LBUS 31-0 contain the data.

TASK 8-15 REQ L         BA30,BC30,BC18-23       Backplane and processor 220/330 ohms
        Open-collector task wakeup request lines.  Protocol is in
        SEQUENCER.DESIGN.  Should be driven by 48 ma open-collector driver
        from a decoder or octal register holding the task assignment.
```

```
LBUS REQUEST L          BC24            Backplane 220/330 ohm
        Asserted by the processor along with an address, to start a memory
        cycle.  The cycle starts at the leading edge of LBUS CLOCK, and the
        request and address go away at the trailing edge.  A new request may be
        started every clock when doing block reads or writes to memory.  For
        random accesses, a new request may be started as often as every other
        clock.

LBUS WRITE L            BC25            Backplane 220/300 ohm
        Modifies LBUS REQUEST L to request a write rather than a read.
        Only valid at the end of the cycle, because processor traps and map
        faults may suppress writes.

LBUS REFRESH L          BC26            Backplane 220/330 ohm
        Asserted by the processor, with the same timing as LBUS REQUEST L,
        to start a refresh in all memories.

LBUS WAIT L             BC27            Backplane and processor 220/330
        Asserted by a device, during the active cycle (see below), to
        stall the processor when the device is slower than normal
        memory.  The active cycle continues for as many clocks as LBUS
        WAIT L is asserted until the first active cycle when it is
        absent.  This signal must be valid early in the cycle (well
        before the leading edge of the clock) and stable until the
        trailing edge of the clock.  It is a tri-state signal driven
        by the device performing the active cycle, and pulled-up by
        the backplane termination (and processor?) when not used.
        [But see later comments about DMA devices asserting wait.]  It
        is used by DMA devices, to gate a state clock for the LBUS.

LBUS RESET L            BC28            Backplane 220/330 ohm
        Asserted by the FEP to reset all devices on the bus.

LBUS POWER RESET L      BC17            Backplane 220/330 ohm
        Asserted by the power supply and/or the FEP to reset all devices on
        the bus when power has come on and not been verified, and also when
        power is turning off.  This is mostly to protect the disk.  LBUS
        RESET L is asserted whenever LBUS POWER RESET L is.  Terminated on
        the backplane.
```

Loading and termination:

Maximum per-card loading requirements on these signals have not yet been established. For the main signals a minimal stub and a single low-current or PNP input is allowed. For the card select address bits, a single 74F521 load is allowed where speed is absolutely required, although a 25LS2521 is preferred.

All address and control signals originating in the processor are driven with 48-64 ma drivers (74F244s or equivalent), and terminated at the far end of the bus on the backplane. Bi-directional signals and signals driven by devices are terminated at the end of the bus and in the processor.

Open collector signals require drivers with 48 ma output capability -- e.g. either Am26S10 quad xcvr, or 74LS641-1, 74LS642-1 octal xcvrs. This means the TASK 8-15 REQ L lines.

Termination that is indicated at the processor might as well be accomplished by SIPs on the processor end of the backplane. Some signals terminate on the MC or SQ boards(?), and the terminations will have to be contained on those boards. [Actually, JVB said that he would put these SIPs on the backplane between the card slots.] Augut Holtites will be used to make the SIPs socketed.

Termination as follows:

	Far end	Other termination
LBUS ADDR 0-23	220/330	-
LBUS 0-35	220/330	270/560 at DP end of bus on backplane
LBUS 36-43	220/330	270/560 on backplane
LBUS REQUEST L	220/330	220/330 on backplane
LBUS WRITE L	220/330	220/330 on backplane
LBUS REFRESH L	220/330	220/330 on backplane
LBUS WAIT L	220/330	220/330 on backplane
LBUS WITH ECC	220/330	220/330 on IFU card
LBUS RESET L	220/330	220/330 at DP end of backplane
LBUS POWER RESET L	220/330	220/330 at DP end of backplane
TASK 8-15 REQ L	220/330	220/330 on backplane at SQ card
LBUS ID REQUEST L	220/330	(sourced at FEP)
LBUS DEV READ L	220/330	(sourced at MC)
LBUS DEV WRITE L	220/330	-
LBUS DEV COND L	220/330	220/330 on backplane at SQ card
LBUS DEV 0-9	220/330	-
LBUS FIRST HALF +,-	68 to -2V	
LBUS CLOCK +,-	68 to -2V	
PROC CLOCK +,-		68 to -2V at DP end of bus
PROC FIRST HALF +,-		68 to -2V at DP end of bus
PROC WP +,-		68 to -2V at DP end of bus
SPY 0-7	--	
SPY ADDR 0-5	--	
SPY DMA SYNC↑	???	
SPY DMA BUSY	???	
SPY READ L	???	
SPY WRITE L	???	
SPY DMA ENB L	???	

(Note: try to isolate SPY bus with XCVR at board edge)

Documentation of Lbus signals

MEMORY AND CLOCK SIGNALS. (From <LMIFU>MC.)

The bus is used in three ways; accessing memory, accessing I/O device registers which look like memory, and accessing "MicroDevices". MicroDevices are distinguished because they are addressed by a separate 10-bit field which comes directly from the microcode, and do not follow the 3 cycle Request/Active/Data protocol of memories. One example of such a device is a DMA device such as the disk; the DMA task microcode commands the disk to put data onto the bus or take it off, while doing a memory cycle. We'll call the three classes of responders "Memory, MemoryDevices, and MicroDevices."

All transactions on the L-bus are synchronous with the system clock. For example, memory responds to requests with a 2 or 3 cycle sequence, viz:

On the first cycle (Request), the processor puts an address on LBUS ADDR, puts the type of cycle on LBUS WRITE, and asserts LBUS REQUEST. All the memory cards compare the high bits of the LBUS address with their slot number. The selected memory card drives the row address onto the RAM address lines, and at the leading edge of LBUS CLOCK starts RAS. After a delay it muxes the column address onto the RAM address lines, and finally at the clock boundary CAS is enabled.

The second (Active) cycle is used to access the RAM: on a read the RAM output is strobed into a latch at the end of the cycle; on a write, the bus has the write data and ECC bits and the RAM WE is driven by a gated Lbus Clock (late write operation). RAS and CAS are reset at the end of this cycle.

During the third (Data) cycle, the latched read data is driven on the bus (during First Half), the RAM chips precharge during their RAS recovery time, and possibly a new Request cycle occurs.

The bus clock is designed so that the memory card can start RAS with the leading edge and start CAS with the trailing edge and be guaranteed of meeting the RAM timing specs. No other use is intended for the leading edge of clock. It is suggested that MemoryDevices initiate response to requests at the trailing edge of clock.

The clock seen by devices on the bus (LBUS CLOCK) is a version of the clock that drives the processor. Its frequency is roughly 5 Mhz but the exact period of each cycle may vary between 180-260 ns depending on the cycle length specified by the microcode. Although the processor controls the cycle length, LBUS CLOCK is unaffected by any clock inhibit conditions in the processor -- operations on the bus proceed independently of the microcode, once they have been initiated. Memory data error-correction will also extend the clock for some period of time.

An exception to this is when the processor takes a trap. In that case LBUS CLOCK is stretched -- the extra time occurs in the second (or high) phase. While the main clock is held high, the clock and sequencer conspire to perform a second cycle internally that fetches the trap handler microinstruction. Because of this, two first-half clocks will happen for only one LBUS CLOCK. If the extended cycle is a Data cycle, the processor will latch the data seen during the first first-half.

Note: The leading edge of FIRST HALF is >>not<< the same as the trailing edge of LBUS CLOCK. First-half is primarily intended as a timing signal that controls enabling data from memories onto the bus. The only other nefarious use you are allowed is to clock something with the mid-cycle edge of FIRST HALF, and then you should be prepared to see two of them on some cycles.

A central Memory Control manages the state of the bus and arbitrates between requests from the processor, IFU, and FEP. Both Memory and MemoryDevices are expected to conform to the same timing protocol. [document FEP/MC arbitration].

Any MemoryDevices (like the TV) that are unable to respond in 3 cycles must assert LBUS WAIT during the Active cycle until they can respond. The memory control state will proceed on the first Active cycle where LBUS WAIT is not asserted. LBUS WAIT should not be present on any other cycle, and must be developed early enough to propogate the length of the bus, go through a xcvr, and gate the clock. DMA devices also watch LBUS WAIT, so they know which cycle is the one that they should read or write the data.

[Perhaps Wait could be asserted by certain DMA devices during the Active cycle of their transfers to give them extra cycles to produce the data. This would mean that Wait would have to be looked at by the memory card -- on read-active cycles it would delay leaving the active cycle, on write-active cycles it would also inhibit the write signal to the RAM until LBUS WAIT is removed. This requires a small change in the control logic of the memory card.]

Block mode operations. In some cases the processor issues a series of requests on back-to-back cycles. This is called "block mode". A new request can be started each cycle. When a block-mode operation in underway, the bus is segmented into a 3-stage pipeline, one stage for addressing, one stage for ram access, and one stage for data transfer (on reads).

The addresses of block mode requests are always in increasing sequential order, although any pattern that avoids referencing addresses [n, n+4] in adjacent cycles would be OK. The existing memory card interleaves on bits 18,1,0, so an individual ram always see at least 4 cycles between requests for sequential locations. Maybe the spec should assume interleaving on just address bit 0, in case some MemoryDevice needs to depend upon that.

MemoryDevices also have to handle block mode requests, because the microcode will not in general want to distinguish references to MOS memory from MemoryDevices. This means that the device must be prepared to accept a request during its "active" cycle. Request cycles are unconditional, there is no way for a device to reject or delay a request. The cycle following a request is the active cycle, which can be repeated (via LBUS WAIT) until the device is ready to accept data (on writes) or enter the data cycle (on reads).

When the MemoryDevice is signaling wait during an active cycle, the processor must delay issuing a new request until the final active cycle.

[Bus interlock signal? The idea is devices that can hang the memory control signal LBUS WAIT LOCK, which means that if the processor is so unfortunate to make a request to the busy device, the device will end up asserting LBUS WAIT. There could be a bit or spec-function in the microcode that says wait until BUS WAIT LOCK goes away before issuing a new request. This would reduce the latency for DMA memory requests. This is only partially effective, since a read cycle to a MemoryDevice that use BUS WAIT will lock the bus in an active cycle until the data is ready. This can't be predicted by the BUS WAIT LOCK signal.]

[add analysis of critical timing of bus protocol, suggested logic for devices]

Most signals are tri-state, driven by 'F244, Am8304B (maybe), or Am298xx which have at least 48 ma of output drive. Signals are terminated on one end of the backplane with 220/330 SIPs which present a low loading of about 20 ma. [They may need to be terminated at both ends.] I see no reason why open-collector xcvrs like Am26S10's can't also be used, although they are likely to have a slower rise time (20-30 ns on the A-machine XBUS).

LBUS <43:0> - Bi-directional data bus, active high tri-state.
LBUS <43:36> are the ECC bits. Driven by processor or FEP on write Active cycles. Driven by memories on read Data cycles. Also used to transfer data between processor and Devices. Also is used to carry the Obus signals from the data path card (E) to the other cards in the processor (I and C). The data bus is 36+8 bits wide because although the current ECC implementation needs only 7 extra bits, most LSI ECC chips require 8 bits, and we may want to use them in the future.

LBUS ADDR <23:0> - Physical address. Tri-state driven from processor or FEP. A physical address of 24 bits is

semi-consistent with allowing a maximum of 31 physical slots, each of which could hold 512K words of memory.
These lines must be driven with 64 mA drivers such as S244; they have more load than the data lines.

LBUS CLOCK +/- -- Differential ECL system clock. Received with a 10125 ECL-TTL translator, which can be wired to produce either polarity of TTL clock. Requires ECL termination network on both ends of these bus signals. This pair of backplane traces should be somewhat isolated from any noise carrying data-path signals. Both edges of this clock are critical.
(There are actually 2 copies of this clock, one for the processor, and one for the Lbus memory and I/O cards.)

LBUS FIRST HALF +/- -- differential ECL timing signal from memory control. Used during Data cycles to enable memory data onto the bus. The memory card drives data onto the bus during the first half of the cycle, the memory control reads the bus data and does error correction. During the second half cycle, the corrected data is driven on the bus from the memory control.
Memories must insure that data is driven out on the bus as soon as possible after the leading edge of FIRST HALF, because the memory control needs most of the first half to decode the ECC syndrome. An LBUS clocked 74F FF output driving the bus driver output enable is recommended.

[For system that have a stand-alone FEP board without the memory control card, it might be convenient to have the memory drive the data on the bus during the full cycle. In this case, the LBUS FIRST HALF signal will be extended.]

LBUS REQUEST L - Request for Memory or MemoryDevices addressed by Bus.Address. Stable by leading edge of Bus.Clock with enough time for address compare and 2 levels of logic.
LBUS REQUEST L and LBUS WRITE L, along with the address, are asserted towards the end of the first cycle of a transaction. The data are transferred during the second or third cycle. The request, write, and address lines are not valid during those cycles (indeed they may be used to start another transaction).

LBUS WRITE L - from the processor or FEP. The write data will be driven onto the bus during the next cycle. Otherwise, the requested cycle is a read, and the memory will drive the bus during the 2nd succeeding cycle.

LBUS WITH ECC - From Memories that don't have ECC bits. Driven during Data cycle. Although this signal is specified as tri-state, it is logically an open-collector signal -- it need only be driven by non-memories, and in those cases need only be driven low. An Am26S10 could be used here if more convenient, although it seems that using the same part used to drive the data lines is often convenient.
Memory devices can ignore this line or pull it up.

LBUS WAIT L - From MemoryDevices. Asserted for as many cycles as necessary to hold memory control in Active cycle state. Must be valid early in the cycle. Same comments about open-collector-ness as previous signal. [But see prior comments about DMA devices asserting wait.]

LBUS REFRESH L - All dynamic RAM memories perform a refresh. All rows of memory refresh at once. The memory array bypass capacitors hold enough charge to supply the RAMs for the refresh cycle, so the transient shouldn't be seen by the power supply. The refresh timer and address counter is in the Memory Control, it has nothing to do with micro-tasking so that the memories will continue to get refreshed when the processor is being single stepped.

LBUS ID REQUEST L - Requests that the selected board supply information about itself. The board selection is by matching LBUS ADDR <23:19> against the slot number (see below). LBUS <7:0> are driven with one of 32 bytes of data selected by LBUS ADDR <6:2>. The format of these data bytes is not yet specified, but generally includes the board type, board serial number, board revision level, and a checksum sensitive to failures of the data and address lines.

Note that memory refreshing may take place, using LBUS ADDR <17:10>, while a board ID is being read using the other address lines. The PROM data should be driven onto the bus for as long as ID REQUEST is asserted. (The memory card is slightly strange in that it "buffers" LBUS ADDR <6:2> through the same latch that it uses to hold the column address during normal memory cycles. This latch is open during LBUS CLOCK, so the memory board doesn't produce correct data until the second cycle after ID REQUEST and LBUS ADDR are present. The FEP compensates for this, and other boards shouldn't necessarily emulate the memory card.

SLOT NUMBERING

LBUS SLOT <4:0> - a slot number built into the backplane. These pins are grounded in a different pattern at each slot; if the board plugged into that slot provides pullups it will see a unique slot number. This is matched against LBUS ADDR <23:19> for Memory, MemoryDevice, and IDRequest operations, and against LBUS DEV <9:5> for MicroDevice operations, to select the desired board. LBUS SLOT <4> is actually bussed across each card cage, and is grounded in the main card cage and left floating in the extension cage. More discussion of this below.

RESET SIGNALS

LBUS RESET L - general reset line. This is brought low when power is turned on, and whenever the FEP feels like asserting it.

LBUS POWER RESET L - brought low when power is not valid. This line is used to protect disks and to perform initializations only needed when first powering on. When the machine is powered up, this line is grounded and remains grounded until the FEP validates the power and cooling and turns it off. This line is also grounded before turning off the power.

MICRODEVICE SIGNALS

LBUS DEV <9:0> - a device address for microdevice operations. Bits <9:5> select a board, by matching against the slot number. The special slot numbers 36 and 37 are used to select the FEP and MC boards, respectively. Bits <4:0> select a register or operation within the board.

LBUS DEV READ L - commands the device to put data onto the Lbus data lines.

LBUS DEV WRITE L - commands the device to take data from the Lbus data lines, at the LBUS CLOCK. Note that when LBUS DEV WRITE is used to inform the device of a DMA memory cycle being started, the Lbus data lines contain unrelated data perhaps associated with an unrelated memory read. LBUS DEV WRITE L should only be depended upon at the clock edge; it should not be used to gate the clock. If the microinstruction doing the microdevice write is NOPed by a trap or by a control-memory parity error (e.g. a microcode breakpoint), LBUS DEV WRITE L will be asserted for a period of time, past the leading edge of the clock, and will then be deasserted some time before the trailing (active) edge of the clock.

LBUS DEV COND L - the selected device may ground this line (with an open-collector nand gate) to feed a skip condition to the microcode.

Microdevice I/O is used for general communication with devices, for internal communication within the processor complex (including the FEP), and for control of DMA operations.

For general communication with devices, the Lbus simply acts as an extension of the processor's internal bus. Data are transmitted within a single cycle and clocked at the trailing edge of the clock.

Microdevice read and write to slot number 36 is used for communication with the FEP, the page tags, and the microsecond clock. Microdevice read and write to slot number 37 is used for communication with the MC and SQ boards. (It is used when reading and writing the NPC register in the SQ board in order to reserve the Lbus and connect it to the datapath; the control signals to the SQ board are transmitted separately.)

DMA works as follows. The device requests a task wakeup when it wants to transfer a word to or from memory. The microcode task wakes up for 2 cycles. The first cycle puts the address on the Lbus address lines, makes a read or write request to memory, and also increments the address. The second cycle decrements the word count, to decide when the transfer is done. The microcode asserts DISMISS during the first cycle (the task switch occurs after the second cycle.) The device is informed of the DMA operation by the microcode through the use of a microdevice write during the first cycle. This microdevice write does not transfer any data to the device, but simply tells it that a DMA operation is being performed, and clears its wakeup request flag. (The wakeup request is removed from the bus immediately, and the flag is cleared at the clock edge.) For a read from device into memory, the device puts the data on the bus during the active cycle (one cycle after the microdevice write) and it is written into memory. For a write, the device takes data from the bus two cycles after the microdevice write.

Some devices look like memory, rather than using microdevice I/O. The criterion for which to use is generally whether the device is operated by special microcode, and the convenience and need for speed of that microcode. Devices that look like memory can be accessed directly by Lisp code.

SPY SIGNALS

SPY <7:0> - an 8-bit, bidirectional, rather slow bus used for diagnostic purposes. Allows the FEP to read and write various cpu state while the machine is running.

SPY ADDR <5:0> - addresses the diagnostic register to be read or written

SPY READ L - gates data from the selected register onto the spy bus.

SPY WRITE L - clocks data from the spy bus into the selected register, on the trailing edge.

SPY DMA SIGNALS

When the spy bus isn't being used for diagnostics, the FEP uses it as a special side-door path to certain DMA devices. Normally the FEP uses it to receive a copy of all incoming network packets; it can also set it up to transmit to the network and to read from the disk (possibly also to write the disk; this is unclear and not yet determined). Details are in <LMHARD>DMA.DESIGN; that part of that file is said to be up to date.

SPY <7:0> - 8 bits of data to or from DMA device. These lines are continuously driven during DMA operations; the FEP's DMA buffer does not latch them.

SPY DMA ENB L - asserted if DMA operations are permitted to take place; deasserted if the spy bus is being used for diagnostic purposes.

SPY DMA SYNC↑ - a clock, asserted by the device. On the rising edge of this a byte is transferred and the address is incremented. The device must take the data (for write) or supply the new data (for read) on or before the leading edge of this. This is the same wire as SPY ADDR 0.

SPY DMA BUSY L - asserted if the DMA operation has not yet completed. This can be asserted by the device or the FEP or both, depending on who determines the length of the transfer. For example, for network input this comes from the device, while for network output and disk input it comes from the FEP (the disk doesn't know it's own block size). This is the same wire as SPY ADDR 1.

Spy Bus Addresses

Updated 11/16/84 Moon
SQ, MC, AU, IO are boards in the old board set.
NBSP refers to the processor board of the new board set [information to be supplied].
VSP refers to the virtual Lbus slot protocol followed by the 2 megaword memory board.

Location	Board	Read	Write
0-15	SQ	UIR<0:111>	CMEM WD<0:111>
16	SQ	SQ Board ID[U AMRA<4:0>]	SQ CTL 1
17	SQ	DP Board ID[U AMRA<4:0>]	SQ CTL 2
20-21	SQ	SQ status	--
22-23	SQ	Next CPC	--
24-25	SQ	SQ status2	--
26-27	SQ	OPC	--
30-37	SQ	(spare)	--
40	MC	MC Board ID[control<4:0>]	MC control
41	MC	MC error status	(spare)
42	MC	Syndrome	--
43	MC	Error address	--
44	MC	MC misc status	--
45	MC	(spare)	--
46	-	(spare)	--
47	AU	AU Board ID[U AU OP<4:0>]	uFPA ENABLE←SPY<0>
50	IO	Net signals	Lbus card select
51	IO	Net status	Net control
52-57 (spare)			
60	VSP	--	Lbus card select
61	VSP	--	<card1slot>_0 + <card2slot>_5(low 3)
62	VSP	--	<card2slot>_-3(high 2) + <card3slot>_2
63-77 (spare)			

The file SYS:L-FEP;SHARED-DEFINITIONS.LIL contains the actual spy bus definitions used by the FEP.

Timing Requirements

LBUS RESET and LBUS POWER RESET are asynchronous. All other side-effects should take place at the trailing edge of the clock. LBUS REQUEST and the address lines are stable before the leading edge of the clock. LBUS WRITE however is only valid at the trailing edge of the clock; it can change as the result of a trap. Consequently it is illegal for memory reads to have side-effects, as memory reads not requested by the program can occur.

In a microdevice write, the address lines (LBUS DEV 0-9) are stable throughout the cycle, however the data (LBUS 0-35) and LBUS WRITE itself are only valid at the trailing edge of the clock. The data lines are only driven by the processor during SECOND HALF.

In a microdevice read, the address lines (LBUS DEV 0-9) are stable throughout the cycle, however LBUS READ itself is only valid at the trailing edge of the clock; side-effects are permitted but may only happen at the clock. The data (LBUS 0-35 or in some devices LBUS 0-31) may be driven on the bus during the entire cycle (when LBUS DEV READ is true) but the data must be valid on the bus before the end of FIRST HALF. It is more conservative timing for the device to enable the data onto the bus with FIRST HALF ANDed with DEV READ. LBUS DEV READ is probably valid on the bus 15ns after clock.

TASK 8-15 REQ and TASK 4 REQ are asynchronous and may be driven at any time. Once a task is requested, it should stay requested until explicitly dismissed or until LBUS RESET. When a task is dismissed, the task request must be deasserted during the cycle that is dismissing, so that a new task of presumably lower priority can be scheduled. The task request flip flop however must not be cleared until the trailing edge of the clock, the time when all side-effects occur. During the cycle after a dismiss the task request will not be looked at by the processor, however the device should deassert its request as quickly as it can (a glitch is expected at the beginning of the cycle).

Data driven onto the Lbus data lines (LBUS 0-43) must be synchronized to the processor clock; failure to observe this rule can cause every sort of internal parity error in the processor as well as memory ECC errors. When reading from memory, the data must be stable on the bus as early as possible, to allow time for the ECC-error decision before the end of FIRST HALF. Memory read data are driven onto the bus during FIRST HALF, and then latched by the processor during SECOND HALF. This latch is followed by a second one, that is opened during the middle of FIRST HALF to pick up the raw data, and again during the middle of SECOND HALF to pick up the ECC-corrected data (if any). ("Middle" is controlled by PROC WP). Even devices that deassert LBUS WITH ECC must provide the data early enough to avoid synchronizer failure in either of these latches. [An estimate of this time is in TMC.DESIGN].

When reading from a microdevice, there is more timing leeway since the microcode knows the specific device it is reading from and can use a slow-first-half cycle. Also there is no ECC computation. The microdevice drives the data lines during the first half and the processor effectively clocks them at the trailing edge of FIRST HALF (actually there is one latch open during FIRST HALF followed by a second latch open during SECOND HALF; this is done for hardware minimization reasons). The device data must be stable early enough to avoid synchronizer failure in these latches. The microcode will use a slow-second-half cycle if necessary, since it does not see the data until SECOND HALF. Lbus data lines not driven by a microdevice will be brought to 1 by the terminator, but not quickly enough to avoid problems. Thus all microdevice reads must drive at least LBUS 0-33. [Hmm, the microsecond clock violates this; I would expect it to cause type-map parity errors. Perhaps the terminator really does provide adequate rise time.]

Note that when doing a memory read, the data are driven two clocks after the request (skipping LBUS WAIT cycles); the bus-driver enable should come from a clocked register. When doing a microdevice read, the data are driven by LBUS DEV READ gated by matching of LBUS DEV ADDR 9-5. LBUS DEV READ takes some time after the beginning of the cycle to become stable, and the device should introduce as little additional delay as it can. The device should only drive the bus during FIRST HALF, so that it turns off in plenty of time before the next cycle.

When writing into memory from a DMA device, the data, including the ECC code added by the memory control, must be stable at the memory chips before the leading edge of the clock (which is when WRITE is asserted to the RAMs). To meet this timing for the ECC bits requires the device to drive the bus by [?? whatever the time is.] Microcode can provide a slow-first-half cycle if necessary.

All sources of data to be driven onto the Lbus, whether for microdevice read, memory-mapped read, or DMA write, should be registers clocked by the Lbus clock. Do not use 74LS244 buffers, or multiplexors, to select asynchronous status signals; run them through a register to synchronize them. In most cases two levels of synchronizer are unnecessary since the value should be stable by the time the bus-driving buffer is turned on; this might need to be examined carefully for individual cases.

When a cycle is extended because of a trap, so that FIRST HALF happens twice, the latch through which the processor receives Lbus data is only opened during the first FIRST HALF. When a cycle is repeated because of LBUS WAIT, memory-read data are only received from the bus during the first instance of the cycle. (This only happens when a block read is done from a device that uses LBUS WAIT, since only in a block read can an active cycle and a data cycle coincide, and LBUS WAIT is associated with active cycles.) Microdevice-write and memory-write data are driven during throughout an extended or repeated cycle (microdevice-write data are only driven during SECOND HALF).

The leading edge of FIRST HALF does not precede the trailing edge of the clock. It is not a good idea to depend on this. The trailing edge of FIRST HALF preceeds the leading edge of the clock.

LBUS WITH ECC is driven with the same timing requirements as the data lines.

LBUS DEV COND must be stable [??] ns before the trailing edge of the clock.

SPY ADDR 5-0 are stable whenever SPY READ or SPY WRITE is asserted. The SPY data lines should be clocked by the trailing edge of SPY WRITE, and should be driven whenever SPY READ is asserted. If a bidirectional transceiver is used to bring the SPY bus onto a board, its direction should be controlled by SPY READ, so that it will not glitch at the trailing edge of SPY WRITE; the FEP latches the SPY lines before it deasserts SPY READ. The FEP allows a long time [?? ns] for a spy read or write, so slow logic may be employed on this bus.

Discussion of card addressing and slot numbers

The idea is that there are no address dip switches anywhere; instead the address of any card depends on what backplane slot it is plugged into. Furthermore you can have multiple copies of a card in a system and they will automatically be at different addresses. This applies both to memory cards and to I/O cards; it does not apply to processor cards, which have their own dedicated backplane slots.

Each backplane slot has 5 pins which are its slot number; they are grounded in a different pattern at each slot, and pulled-up on the card.

Each memory card, and other memory-like device such as a TV, responds to the physical address which corresponds to its slot number in bits 23:19. This provides for 512K memories. Smaller memories only work at some subset of their addresses. Larger memories will have to tie up multiple slots somehow.

I/O cards also respond to such addresses as if they were memories, if they are Memory-Mapped devices. (The previous idea of packing all the I/O cards into the address space for slot 37 has been flushed). Microdevice devices have a separate address space using the LBUS DEV <9:0> lines, but still use the slot number to tell whether they are selected.

The LBUS ID REQUEST L signal, in combination with the address and data lines, allows a 32x8 PROM in the card whose slot number matches LBUS ADDR <23:19> to be read by the FEP. This makes it possible to determine what kind of card is present in that slot and what options it has. When the machine is booted, the configuration of memory and I/O cards is automatically discovered and tables of the addresses of the important devices are set up. Either this is done by the diagnostic microprocessor before trying to boot the Lisp engine, or the basic devices required to run Lisp code (some memory and the boot disk) have to be in fixed slots, and the remaining configuration is determined by Lisp code. It may be a requirement that slot 0 always contain a memory card if there are any wired-in physical memory addresses. (It would be a good idea to avoid this.) Memory cards can be depopulated, so the size of each memory card is also automatically discovered during booting. The FEP will also run memory diagnostics to discover the location of any hard failures, which will be patched out, and will initialize the memory to have good ECC. When the FEP loads the microcode, it patches in slot numbers of devices where necessary, so that the microcode will work no matter how the cards are plugged in on the particular machine.

In general it is not possible to move the cards around randomly, because they have I/O connectors on the backplane. However it is desirable to have the card locations be flexible, to be able to have more than one of each device, and to be able to move the memories around randomly.

In some cases, more than one device are combined onto one card. For example, the disk and TV are planned to be on the same card. The device register space is partitioned between the combined devices as necessary, and the ID prom identifies the combination to the system.

TV cards use part of their address space for the video buffer memory, and part for any necessary control registers.

The microdevice addresses for slot 37 are used by the memory control.
The microdevice addresses for slot 36 are used by the FEP.

Processor section of backplane

Signals are listed in pin order. Each signal includes the pin on which it appears and the cards to which it is connected. A * follows cards that can drive the signal. A - follows cards that are connected to the signal but don't use it for anything. BUS as a card means that the signal extends out of the processor and is bussed to all Lbus slots. In general all signals are bussed straight across through some contiguous segment of the processor backplane. When this is not the case, it is noted explicitly.

Note that the C and D connectors on the FEP are mostly free for connection to a paddleboard. Some signals at the top of the C connector are used to control the clock, which resides on the FEP board. [There may also be some memory-control signals here; I'm not sure yet.]

Signals that are not documented here are Lbus signals documented above or microcode bits, documented in MICROINSTRUCTION.BITS.

This backplane differs somewhat from the backplane of the first prototype. The next page documents the differences. (The archetype machine will eventually be modified to conform.)

The order of processor cards, from left to right as seen from behind the backplane (i.e. the pin side), is:

```
        DP - data paths
        SQ - sequencer, control memory
        MC - memory control, instruction fetch unit
        AU - floating point Arithmetic Unit option
        FEP - front-end processor
        BUS - the bus extends from here over
        There are also terminators to the left of DP.
        If no AU is present, the slot must be left vacant.
                [Old idea of plugging a memory into it has been flushed,
                because the power pinout at this slot must be different.]
```

LBUS ADDR 0-11	AA1-12	DP SQ- MC* AU- FEP* BUS
LBUS ADDR 12-23	AA13-24	MC* AU- FEP* BUS
U TYPE MAP SEL 0-5	AA13-18	DP SQ*
SPY READ DP ID L	AA19	DP SQ*
U XYBUS SEL	AA20	DP SQ*
U STKP COUNT	AA21	DP SQ*
U OBUS CDR 0-2	AA22-24	DP SQ*
U OBUS HTYPE 0-2	AA25-27	DP SQ*
LBUS ID REQUEST L	AA25	MC- AU- FEP* BUS
LBUS BLOCK REQUEST L	AA26	MC* AU- FEP- BUS-
LBUS DEV READ L	AA27	MC* AU- FEP BUS
U OBUS LTYPE SEL	AA28	DP SQ*
LBUS DEV WRITE L	AA28	MC* AU- FEP BUS
LBUS DEV COND L	AA29	DP- SQ MC- AU- FEP- BUS*
FEP CONTINUITY	AA30	DP SQ MC AU FEP*

Asserted by the FEP and read back on the other continuity lines to detect the presence of processor boards (and in the correct slots).

MC CONTINUITY	AA31	DP- SQ- MC* AU- FEP

Jumpered to FEP CONTINUITY on the MC card.

SQ CONTINUITY	AA32	DP- SQ* MC- AU- FEP

Jumpered to FEP CONTINUITY on the SQ card.

LBUS 0-29	AC1-30	DP* SQ MC* AU FEP* BUS*

```
DP CONTINUITY           AC31           DP* SQ- MC- AU- FEP
        Jumpered to FEP CONTINUITY on the DP card.

AU CONTINUITY           AC32           DP- SQ- MC- AU* FEP
        Jumpered to FEP CONTINUITY on the AU card.

SPY 0-7                 BA1-8          DP- SQ* MC* AU* FEP* BUS*

SPY ADDR 0-5            BA9-14         DP- SQ MC AU FEP* BUS
        SPY ADDR 0-1 also used for FEP-DMA

SPY READ L              BA15           DP- SQ MC AU FEP* BUS

SPY WRITE L             BA16           DP- SQ MC AU FEP* BUS

SPY DMA ENB L           BA17           FEP* BUS

(spare)                 BA17           DP- SQ- MC- AU-

TASK 4 REQ L            BA18           DP- SQ MC- AU- FEP- BUS*
        Low-priority task wakeup

LBUS DEV 0-9            BA19-28        DP SQ* MC AU- FEP BUS
U AMWA 0-9
        Note that these lines have two names, since they serve as both the
        Lbus microdevice address and some datapath control signals.  The same
        wires are bussed all the way through both the processor and the Lbus.

LBUS FIRST HALF +,-     BA29,BC29      FEP* BUS
        Terminate with 68 ohms to -2V at end of BUS.

(spare)                 BA29,BC29      DP- SQ- MC- AU-

TASK 8-9 REQ L          BA30,BC30      DP- SQ MC- AU- BUS*
        (See below; listed here since they fall here in pin order)

(spare)                 BA31           DP- SQ-

-COND                   BC31           DP* SQ*

EXTERNAL REQUEST L      BA31           MC- *** BUS*
EXTERNAL GRANT L        BC31           MC* *** BUS-
        Traces between SQ and MC should be cut. These will have
        to be jumpered around the AU and FE slots.

LBUS CLOCK +,-          BA30,BC30      FEP*
                        BA32,BC32      BUS
        Terminate with 68 ohms to -2V at end of BUS.
        Note that these signals change pin number at the FEP.

PROC CLOCK +,-          BA32,BC32      DP SQ MC AU
                        BA31,BC31      FEP*
        Separately-driven duplicate of LBUS CLOCK.
        Terminate with 68 ohms to -2 V at DP end.
        Note that these signals change pin number at the FEP.

LBUS 30-35              BC1-6          DP* SQ MC* AU FEP* BUS*

LBUS 36-43              BC7-14         MC* AU FEP* BUS*

DP TRANSPORT TRAP L     BC7            DP* SQ
        Asserted if a trap is required for garbage-collector processing
        of the data being read from memory (a function of the data type
        and the high-order address field).

DP TYPE TRAP            BC8            DP* SQ
        Asserted if the type map calls for a trap (bad data type or
        invisible pointer).

DP TRAP PARAM 0-3       BC9-12         DP* SQ
        Trap parameter (dispatch code for arithmetic trap, trap number
        for type trap).

DP SLOW JUMP L          BC13           DP* SQ
        Asserted if a non-NOPing trap is required (used by the stack
        garbage collector that doesn't exist yet).
```

```
DP MISC TRAP            BC14            DP* SQ
        IOR of trap conditions other than the above.

LBUS WITH ECC           BC15            MC AU- FEP BUS*

AMEM PAR ERR L          BC15            DP* SQ
        Parity error in A-memory; stops machine

(spare)                 BC16            DP- SQ- MC- AU- FEP- BUS-
        Spare Lbus line

LBUS POWER RESET L      BC17            DP SQ MC AU FEP* BUS
        Terminate somehow.  May need to be brought out to power supply?
        (May go to front panel also, but FEP will provide that connection.)

TASK 8-15 REQ L         BA30,BC30,BC18-23       DP- SQ MC- AU- FEP- BUS*
        TASK 8-9 REQ L are not connected to the FEP.

LBUS REQUEST L          BC24            MC* AU- FEP* BUS

TYPE PAR ERR L          BC24            DP* SQ
        Parity error in type map

LBUS WRITE L            BC25            MC* AU- FEP* BUS

GC MAP PAR ERR L        BC25            DP* SQ
        Parity error in garbage-collector address-space-quantum map

LBUS REFRESH L          BC26            MC- AU- FEP* BUS

BMEM PAR ERR L          BC26            DP* SQ
        Parity error in B-memory; stops machine

LBUS WAIT L             BC27            DP SQ- MC AU- FEP BUS*

LBUS RESET L            BC28            DP SQ MC AU FEP* BUS

PROC WP +,-             CA1,CC1         DP SQ MC AU FEP*
        Write-pulse for internal static RAMs; occurs twice per cycle.
        Terminate with 68 ohms to -2 V at DP end.

PROC FIRST HALF +,-     CA2,CC2         DP SQ MC AU FEP*
        Separately-driven duplicate of LBUS FIRST HALF.
        Terminate with 68 ohms to -2 V at DP end.

CLK EXTEND CYCLE        CA3             DP* SQ- MC* AU- FEP
        A wired-OR ECL signal, asserted when extra time is needed for a trap.
        Terminate with 100 ohms to -2 V at DP end and on FEP.

CLK CS PRESET L         CA4             DP SQ- MC- AU- FEP*
        Forces chip-select for A,B memories on at the beginning of the cycle,
        until there has been enough time for the pass-around decision.
        (Saves a few nanoseconds).

SQ NEXT INST L          CA5             DP SQ* MC AU- FEP-
        Asserted if this is the last microinstruction for this
        macroinstruction.

U AMRA 0-5              CA6-11          DP SQ*

FEP LBUS RQ L           CA6             MC AU- FEP*
        Asserted if FEP wants the bus or is using it (active cycle).

REFRESH RQ L            CA7             MC AU- FEP*
        Asserted if time for a memory refresh, or refresh active cycle.

MC ECC DELAY            CA8             MC* AU- FEP
        Extends the clock during the second half in order to provide
        time for single-bit error correction.
        This is an ECL signal.

DOUBLE ECC ERROR L      CA9             MC* AU- FEP
        True if there is an uncorrectable error in the data for this
        memory read.

(unknown)               CA10-11         MC AU- FEP
```

```
        Signals between MC and FEP not detailed yet

U AMRA 6-11              CA12-17        DP SQ* MC AU(-?)

U AMRA SEL 0-1           CA18-19        DP SQ* MC AU(-?)

U AMWA 10-11             CA20-21        DP SQ* MC AU(-?)

U AMWA SEL 0-1           CA22-23        DP SQ* MC AU(-?)

U MAGIC 0-3              CA24-27        DP SQ* MC AU

U SPEC 0-4               CA28-32        DP SQ* MC AU

CLK WD ENB L             CC3            DP SQ- MC- AU- FEP*
        Another timing signal for A,B memory.

DP SET GC TAG L          CC4            DP* SQ- MC- AU- FEP
        Registered output from the GC map indicating that the
        Abus datum is a pointer to a temporary space.  This sets
        a GC page tag bit if main memory is being written.

NOP L                    CC5            DP SQ* MC AU FEP-
        Asserted if the current microinstruction should not do
        anything, because the processor is stopped, stalled, or
        trapping (valid late, should not be used to gate the clock).

U SPEED 0-1              CC6-7          DP- SQ* MC- AU- FEP

CLK EXTRA INNINGS        CC8            DP- SQ MC- AU- FEP*
        Asserted during the second cycle of a trap.

TASK 3 REQ               CC9            DP- SQ MC- AU- FEP*
        Task wakeup from the FEP

MC PROC NORMAL GRANT L   CC10           DP SQ- MC* AU- FEP
        Asserted if the LBUS ADDR lines contain an address derived
        by mapping the VMA to a physical address.  This signal enables
        the DP card to capture the mapped address for possible later
        use in addressing A-memory.  Also used by the page tag memory.

PAGE TAG PAR ERR L       CC11           DP- SQ MC- AU- FEP*
        Parity error in page tag memory; stops machine.

SPARE ERROR L            CC12           DP- SQ MC- AU-
        Grounding this halts the machine after completing the current
        microinstruction; mostly provided to ease construction of
        temporary debugging kludges.

(spare)                  CC13-15        DP- SQ- MC- AU-
        Bus these across processor (except FEP) and maybe we'll
        find a need for them.

INST 0-7                 CC16-23        DP MC*
        Low 8 bits of the current macroinstruction.
        Note: these lines are wired around the SQ slot.

U AU OP 0-7              CC16-23        SQ* AU
        Microcode control for the AU.
        [This assumes 8 more bits of control memory are wedged in.]
        Note: these lines are wired around the MC slot.

AU STOP L                CC24           SQ AU*
        Any error on the AU that needs to stop the mchine.
        Note: this line is wired around the MC slot.

(spare)                  CC25-28        SQ- AU-
        Connect these between the SQ and AU for possible future use
        Note: these lines are wired around the MC slot.

SEQUENCE BREAK           CC24           DP* MC
        Macrocode interrupt request.
        Note: this line is wired around the SQ slot.

MC COND                  CC25           DP MC*
        A microcode skip condition.
        Note: this line is wired around the SQ slot.

MC OBUS TO LBUS L       CC26            DP MC*
        Enables the datapath output to drive the Lbus
        Note: this line is wired around the SQ slot.

MC OBUS REG TO LBUS L   CC27            DP MC*
        Enables the datapath result from the previous microinstruction
        to drive the Lbus (used when writing main memory)
        Note: this line is wired around the SQ slot.

MC ADDR IN AMEM L       CC28            DP MC*
        Indicates that the VMA maps to an A-memory address
        Note: this line is wired around the SQ slot.

MC ABUS 32-35           CC29-32         DP* SQ- MC* AU*
        Data bus between DP, MC, and AU.

MC ABUS 0-31            DC1-32          DP*
                        DA1-32          MC* AU*
        Bidirectional data bus between DP, MC, and AU.
        Note: this is wired around the SQ slot.
        Note: this is on the "C" column at the DP, but the "A" column
        elsewhere.

        [??? does the MC Abus need termination???]

U BMRA 0-7              DA1-8           DP SQ*

U BMWA 0-3              DA9-12          DP SQ*

U BMEM FROM XBUS        DA13            DP SQ*

U COND FUNC 0-1         DA14-15         DP SQ*

U COND SEL 0-4          DA16-20         DP SQ*

U BYTE F 0-1            DA21-22         DP SQ*

U ALU 0-3               DA23-26         DP SQ*

DISPATCH 0-3            DA27-30         DP* SQ
        Contents of field being dispatched on

(spare)                 DA31-32         DP- SQ-
        I guess these are bussed

(spare)                 DC1-4           SQ- MC-
        I guess these are bussed

CUR TASK 0-3            DC5-8           SQ* MC
        Task in which the current microinstruction is executing

TASK SWITCH L           DC9             SQ* MC
        Asserted if the next microinstruction will be from a different task

WANT NEXT INST          DC10            SQ* MC
        Asserted if the address supplied by the IFU in the previous cycle
        is actually being used as the next microinstruction address.
        Stalls the processor if the address was not valid after all.
        [Logically the same as SQ NEXT INST L but much faster and not
        gated by NOP.]

MC WAIT                 DC11            SQ MC*
        Asserted if the processor must stall and wait for the Lbus

MC MAP MISS L           DC12            SQ MC*
        Asserted if a map-miss trap should be taken

MC TRAP PARAM 0-1       DC13,14         SQ MC*
        Modifiers for trap address

MC TASK INHIBIT L       DC15            SQ MC*
        Inhibits a task switch after the next instruction.

MC STOP L               DC16            SQ MC*
```

```
        Any parity error on MC board; stops processor.

IFU DISP 2-13           DC18-28         SQ MC*
        Control-memory address of the first microinstruction to execute
        the next macroinstruction

(spare)                 DC29-30         SQ- MC-

U MEM 2-0               DC17,DC31-32    SQ* MC
        Memory-control control field
        Bit 2 is not next to the other bits for historical reasons


Pins DC1-32 on the AU slot are left unconnected for possible cabling
to a second board or other expansion.

Pins CA11-32, CC12-32, DA1-32, DC1-32 on the FEP slot are left unconnected
for paddleboard use.
```

Changes from first-prototype backplane

The U MEM field has been increased from 2 to 3 bits.

LBUS WAIT L (BC27) now runs the full length of the processor part of the backplane. The DP board needs it.

AMEM PAR ERR L moved from BC27 to BC15 (because of LBUS WAIT L change).

Flush UNCORRECTABLE MEM ERR L, MAP PAR ERR L, GC TAG PAR ERR L.
Add MC STOP L, AU STOP L, PAGE TAG PAR ERR L.

Flush HALT L (Lbus arbitrator can be made to not use it).

Move PROC CLOCK +/- to same pins as corresponding LBUS signals. Keep them on the same pins on the FEP board since they are still driven separately.

Move -COND from BC32 to BC31 (because of PROC CLOCK move).
Get rid of COND (formerly BA32).

-5.2 and -2.0 power supply pins moved. Be careful!!!!

Flush TASK 4 REQ SYNC (CC10). Add TASK 4 REQ L (BA18) which runs down the Lbus, providing a low-priority wakeup from random devices (all devices on the same task).

Flush GC TAG COND.

Replace GC TEMP L (CC25) with DP SET GC TAG L (CC04). Same signal except passes through a register.

U AMRA <5:0> no longer go to MC. These pins used for MC/FEP communication.

Move MC PROC NORMAL GRANT L from CC15 to CC10 since it needs to go to the FEP for the page tag memory.

TASK 3 REQ SYNC replaced by TASK 3 REQ; may as well synchronize it on the SQ board.

Some new signals between MC and FEP, and some new signals between MC and SQ.

Move LBUS DEV 0-8 up by one pin, and put LBUS DEV 9 adjacent to them.

Move LBUS POWER RESET L from BC30 to BC17.

Move TASK 8 REQ L from BC16 to BA30, but route it around the FEP slot.
Move TASK 9 REQ L from BC17 to BC30, but route it around the FEP slot.

Add the AU slot (formerly known as FPA).

```
 Recent updates -- 4:42pm  Monday, 15 November 1982
Signal LBUS BLOCK REQUEST L is added to AA26.  Source is MC.
```

```
-*- Mode: Text -*-
G:>Info>

This directory is a repository of general information about the 3600 and
pointers to same.  Contributions are welcome.

** NOTE!! **

I would like to discourage the unnecessary duplication of files on
Godzilla.  If there is a file which you feel belongs in this directory,
but already exists elsewhere on G, CREATE A LINK TO IT.  Links take up
very little space, and insure that anyone who looks here gets the
current version.

-- Chucko
```

3600 ARCHITECTURE

The following pages consist of a loosely-organized discussion of various aspects of the 3600 architecture, as they relate to troubleshooting the system hardware.

These topics are discussed:

I hope to add to this in the future, specifically on the subjects of 'what happens when the machine is booted,' 'more about macroinstructions,' and more 'fun with LCONS.'

600 ARCHITECTURE

MACROCODE vs. MICROCODE

One of the fundamental design features of the 3600 processor is that it is a micro-programmed machine. Therefore, when debugging the processor it is essential to understand the difference between microcode and macrocode:

Microcode is stored in the Control Memory (CMEM) on the sequencer board. The micro-instruction word is 112 bits wide, and CMEM stores 8K words. Microinstructions are fetched from CMEM and executed at the rate of one per processor clock cycle (PROC CLOCK). When the microinstruction is fetched from CMEM, it is latched in the Microinstruction Register (UIR). The outputs of the UIR control the various functions of the 3600 hard-ware during that clock cycle. Part of the microinstruction word is used to determine the address of the next word to be fetched from CMEM, and thus controls the sequence of events in the machine.

Macrocode can be thought of as the 'instruction set' or 'assembly language' of the 3600. Macrocode is storedin virtual memory and is closely related to LISP. Macrocode is not executed directly, but rather, each macroinstruction calls one or more microinstructions, which perform the actual hardware functions.

The 'translation' of macrocode into microcode occurs in the following manner:

1) The microcode initiates LBUS read cycles to fetch four macroinstructions from main memory. The macroinstructions are latched in instruction buffer (IBUF) registers on the MC board.

2) A portion of the current macroinstruction in IBUF is gated onto the IFU DISP lines to the sequencer. The value on these lines is used as the next CMEM address. This CMEM address is called the microcode entry-point for that macroinstruction.

3) A series of one or more microinstructions performs the function specified by the macroinstruction.

4) When this series of microinstructions is complete, the next macroinstruction in IBUF to be gated onto the IFU DISP lines, and the process repeats. When all four instructions in IBUF have been executed, the process goes back to step 1).

One of the confusing things about macrocode and microcode in the 3600 is that the LCONS de-bugging software uses very similar mnemonics for micro- and macroinstructions. For example, the symbol PUSH-IMMED 1 represents a macroinstruction, and PUSH-IMMED represents the microinstruction (actually the CMEM address of the microinstruction) that performs this macroinstruction. The significant difference is where each is found in the 3600: Macroinstructions are stored in virtual memory (@V or @L), whereas microinstructions are stored in CMEM (@C) and are referenced by the OPC, CPC, and NPC registers on the sequencer.

Two other points about macrocode and microcode:

1) The way that the microcode actually branches to the entry-point of the next macroinstruction is by doing a Popj function when CSP = 17.

2) The microcode routine that 'refills' the IBUF registers with four new macroinstructions is called an IFU-EMPTY-TRAP sequence.

(The above two comments will probably make very little sense if you do not already have some familiarity with the 3600 microcode. Consider them here for future reference.)

3600 ARCHITECTURE

MACROINSTRUCTION FORMAT

Macroinstruction words are 17 bits long, and are stored two per 36-bit data ~~word~~
in virtual memory, in the following format:

```
                      Virtual Memory Location

   35   34  33  32 31                    16 15                         0
   -----------------------------------------------------------------------
   |   |   |      | odd instr.  bits 15 - 0 | even instr.  bits 15 - 0     |
   -----------------------------------------------------------------------
     :   :     :
     :   :   not used
     :   :
     :  even instruction bit 16
     :
     :
 odd instruction bit 16
```

The two instructions in each virtual memory word are called 'odd' and 'even.' Note
that during macrocode execution, the even instruction is always executed first.

Each 17-bit macroinstruction can be one of two general types: operand and no-operand.
The format of each is as follows:

```
                          Operand Instruction

             16 15                    8 7                    0
             -----------------------------------------------
             |  |   operand bits 7 - 0  |  opcode bits 7 - 0    |
             -----------------------------------------------
              :                                  :
              :                        (these bits do not equal 377)
          opcode bit 8
                                          opcode bit 9 =0


                          No-operand Instruction

              16                      8 7                      0
              -----------------------------------------------
              |  opcode bits 8 - 0     |  1  1  1  1  1  1  1  1  |
              -----------------------------------------------
                                                 :
                                            (set to 377)

                                             opcode bit 9 = 1
```

In both cases, the opcode field is used to generate the CMEM address for the microcode
entry-point. (Note that opcode bit 9 is the logical AND of instruction bits 7 - 0.)

The operand field of the first type is used to supply immediate data or addresses
to the hardware.

NOTE: The MC logic actually shifts the opcode two bits to the left to generate
IFU DISP, which becomes the CMEM address: opcode bits 9 - 0 become CMEM address
bits 11 - 2.

For example, the PUSH-IMMED 1 macroinstruction looks like this:

```
                        PUSH-IMMED 1
           16   15                 8 7               0
           -------------------------------------------
```

```
        | 0 |  0 0 0 0 0 0 0 1  |  0 1 0 0 0 0 1 1  |
        ---------------------------------------------
          :          :               :
          :      operand = 1         :
          :                          :
          \..........................:
                                     :
                               opcode = 0103
```

The microcode entry-point for this instruction is 0414@C (0103 shifted left two bits). The operand value of 1 is the immediate data that gets pushed on the stack. (More about the stack later.)

A full list of macroinstructions and opcodes can be found in the file >lmach>Opdef.lisp.

PROGRAM COUNTER

The macrocode program counter (PC) is a 36-bit hardware register located on the MC board. (Not to be confused with the microprogram CPC, NPC, and OPC registers!) The PC has the following format:

```
                          Program Counter (PC)

       35  34 33  32 31 30         28 27                                  0
       -------------------------------------------------------------------
       | 0 0 | 1  1 |  | 0  0  0  0 |    virtual address of macroinstr.   |
       -------------------------------------------------------------------
                       :
                       :
                     0 = even instruction
                     1 = odd  instruction
```

Bits 27 - 0 indicate the virtual address of two macroinstructions, and bit 31 indicates whether the odd or even instruction is currently being executed. Another way to think of it is that bit 31 is the least-significant bit of the PC. The 3600 uses bit 31 rather than 0 in order to be compatible with the rest of the memory addressing hardware in the machine.

As mentioned before, the even instruction is executed before the odd one. The following example shows the sequence of PC values that would occur when executing five instruction starting at loacation 34170 in virtual memory:

```
                140000034170     ;even instr.
                160000034170     ;odd instr.
                140000034171     ;even instr.
                160000034171     ; etc.
                140000034172     ; etc.
```

3600 ARCHITECTURE

FUN WITH LCONS (part 1)

The following is a little exercise that illustrates the concepts behind macrocode and the PC. This requires a reasonably healthy 3600 connected to LCONS.

1) Reset the 3600 and do the following at the 3600 console:

```
clear machine
load microcode
load world >world2.load
set chaos 25410 (or whatever your local chaos address is)
```

2) At LCONS type the following: (Note that <return> means 'hit the return key,' <alt> means 'hit the alt mode key',etc.)

```
:faster-fep<return>
<alt><alt>v  pc<alt>v cpc/<alt>v=<alt>v
push-immed<alt>b
```

This sets a breakpoint at the push-immed microinstruction, and causes the PC, CPC mnemonic, and the numeric value of the CPC to be displayed in the LCONS status window.

3) At LCONS type:

```
:start<return>
```

The 3600 should start and then stop. (If it doesn't stop, hit the space bar.) The LCONS status should now contain, amoung other things:

```
CMEM-parity-error (caused by the breakpoint)
.
.
PC/  160000034160
CPC/  PUSH-IMMED
CPC= 414
```

4) At LCONS type: (Things inside the { } brackets are reponses from LCONS.)

```
pc/  { 160000034160 }   ;q   { CDR-NEXT DTP-ODD-PC 34160 }   <tab>
{ 34160@v/ 40120600420}    <alt>;i    { nil?? ≵ PUSH-IMMED 1 }        <line>
{ 34161@v/ 40020600111}    <alt>;i    { POP-INDIRECT @XX|0 ≵ PUSH-IMMED 0}   <line>
{ 34161@v/ 40124000511}    <alt>;i    { POP-INDIRECT @XX|1 ≵ PUSH-N-NILS 1}
```

This shows that the PC points to the odd instruction at address 34160. We have also displayed several macroinstructions in virtual memory.

NOTE NOTE NOTE !!!!!

LCONS displays the even macroinstruction to the LEFT of the ≵ and the odd macroinstruction on the RIGHT!!!! This is the reverse of how the instructions are stored in memory. However, it does mean that the instruction displayed on the left is the one that gets executed first.

The current macroinstruction in the above case is PUSH-IMMED 1, and its microcode entry point is 414@C, i.e., the PUSH-IMMED microinstruction.

5) At LCONS type c-n (control-n) to single-step the 3600 by one clock cycle. The status window now shows:

```
PC/    140000034161
CPC/    IFU-EMPTY-TRAP
CPC =   14000
```

In other words, the 3600 has completed execution of the PUSH-IMMED macroinstruction, and has incremented the PC (34161, even instruction). The IBUF registers on the MC board are apparantly empty, because the microcode has branched to the IFU-EMPTY-TRAP instruction, which is perfectly normal.

FUN WITH LCONS (part 1, cont'd)

6) Type c-n four more times to complete the IFU-EMPTY-TRAP sequence. (The PHYS-MEM-READ-DELAY microinstruction is part of this sequence.) The status window should now show:

```
PC/   140000034161
CPC/   POP-INDIRECT
CPC= 444
```

In other words, the 3600 is about to execute the POP-INDIRECT @XX|10 macroinstruction, which has its microcode entry-point at 444@C. POP-INDIRECT is the name of the microinstrction at 444@C.

7) Continue to single-step the 3600 using c-n. Notice that unlike PUSH-IMMED 1, POP-INDIRECT @XX|10 takes several microinstructions to complete its execution. If you single-step long enough, the IBUF registers will become "empty" again and the microcode will branch to another IFU-EMPTY-TRAP sequence.

STACK and STACK POINTER

The 3600 processor uses what is referred to as a stack architecture. This means that all arithmetic, logical, and data-movement macroinstructions use a stack rather than general-purpose registers. Unlike such processors as the 8080, 68000, or PDP-11, the 3600 has no register-to-register instructions.

NOTE

- The stack being discussed here is the macroinstrcution, or LISP, stack. This is not to be confused with the microinstruction stack and stack pointer (CSTK, CTOS, CSP).

For example, the following 3600 code adds the values 1 and 2 and stores the result in virtual memory:

```
PUSH-IMMED 1                ;push the value 1 on the stack
PUSH-IMMED 2                ;push the value 2 on the stack
BUILTIN +-INTERNAL STACK    ;add the top two values on the stack and
                             leave the result on the top of the stack
POP-INDIRECT @XX|2          ;pop the value off the top of the stack
                             and store in virtual memory
```

From a software point of view, the stack resides in virtual memory. However, for the sake of speed, the system makes sure that the top area of the stack is always mapped into A-memory on the DP board. In other words, the top several (up to 1k?) locations of the stack always physically reside in AMEM.

In conjunction with this, there is a hardware register on the DP board called the Stack Pointer (SP). This a 28-bit register that can be thought of as containing the virtual address of the top location of the stack. However, the low-order 10 bits can be gated directly onto the AMEM addres lines, so that the processor can access the top of the stack with a single microinstruction, and without going through the virtual address mapping procedure.

The following example illustrates the changes that the SP and the stack would go through when executing the previous four lines of code. Note that by convention the stack is illustrated in 'push-down' format, i.e., the top of the stack moves down the page when pushing.

1) Before executing the first instruction:

```
                                              AMEM

         SP                           4024 |              |
  -----------------                        -----------------
  | 1777774025    | ------------------> 4025 |  40000000011 |  ;some previous value
  -----------------                        -----------------   is at top of stack
                                      4026 |              |
```

2) After executing PUSH-IMMED 1:

```
                                              AMEM

         SP                           4024 |              |
  -----------------                        -----------------
  | 1777774026    | ----------:         4025 |  40000000011 |
      -----------------          :          -----------------
                                 -------->  4026 |   40000000001 |  ;ucode adds CDR and
                                                  -----------------   type bits to immed.
                                            4027 |               |   value & pushes it
```

STACK and STACK POINTER (cont'd)

3) After executing PUSH-IMMED 2:

```
                                                   AMEM

          SP                           4024 |                  |
   -----------------                        --------------------
   |  1777774027   |  -----------:     4025 |   40000000011    |
   -----------------             :          --------------------
                                 :     4026 |   40000000001    |
                                 :          --------------------
                                 --------> 4027 |   40000000002    |   ;2nd immed. value
                                            --------------------      pushed onto stack
                                       4030 |                  |
```

4) After executing BUILTIN +-INTERNAL STACK:

```
                                                   AMEM

          SP                           4024 |                  |
   ----------------                         --------------------
   |  1777774026  |  ---------:         4025 |   40000000011    |
   ----------------           :             --------------------
                              ----------> 4026 |   40000000003    |   ;top two values are
                                            --------------------      popped, then added,
                                       4027 |                  |      result is pushed
```

5) After executing POP-INDIRECT @XX|2:

```
                                                   AMEM

          SP                           4024 |                  |
   ----------------                         --------------------
   |  1777774025  |  --------------------> 4025 |   40000000011    |   ;result is popped
   ----------------                         --------------------       off the stack and
                                       4026 |                  |       stored in virtual
                                                                        memory, original
                                                                        value is back
                                                                        on top of stack
```

The exact location of the stack in AMEM varies dynamically, but when the 3600 is booted, the top of stack will start off around 4000@A.

The DP board also contains the hardware to increment or decrement the SP, and to specify an offset of up to plus or minus 128 decimal when addressing AMEM with the SP (AMRA-SEL or AMWA-SEL = Base+offset). In addition, the microcode maintains a copy of the top location of the stack at address 360 in B-memory. Thus when the microcode adds the top two words on the stack, it gets one argument from 360@B and the other from AMEM using SP with an offset of -1 as the address. The result is stored both in 360@B and (SP)-1@A, and finally the SP is decremented. Through the miracle of modern microprogramming, all this takes place during on micoinstruction cycle. The applicable microinstruction fields would be something like this:

```
AMRA-SEL: Base+offset
AMRA: 377  (base = SP, offset = -1)
BMRA: 360
XYBUS-SEL: A→X,B→Y
ALU: X+Y (In real life, the ucode uses ALU: X+1 and enables 'weird ALU functions',
                which really means X+Y+overflow. This is done in order to totally
                confuse the uninitiated.)
AMWA-SEL: Base+offset
AMWA: 377 (base = SP, offset = -1)
BMWA: 0  (which means 360@B)
Count STKP (AMWA bit 11 = 0, so SP is decremented.)
```

FRAME POINTER

The Frame Pointer (FP) is another 28-bit register on the DP board, and is used in conjunction with the SP. The FP can also be used to address AMEM, with a poistive or negative offset if desired, but cannot be incremented or decremented by a single microinstruction.

The FP is used to mark a sort of 'working area' in the stack. To be more precise, the macrocode is grouped into what are called compiled functions. Each compiled function corresponds to a specific LISP function. During the time each compiled function is being executed, all stack operations take place within a specific area of the stack called the stack frame. The FP points to the lower boundary of the frame. When a new function is called, the PC, SP, FP, and are saved on the stack. Arguments to be passed to the new function are also put on the stack and a new stack frame is created, and the FP is changed accordingly. The structure of a stack frame is shown below.

```
                                            AMEM
                                     |                 |
                                     -------------------
                                     |                 |   --- top of previous stack frame
                                     -------------------
                                     |    Old FP       |   ---:
                                     -------------------      :
    ↓ Increasing Addresses ↓         |    Old SP       |      :-- used to return to previous
                                     -------------------      :   function
                                     |    Old PC       |   ---
                                     -------------------
                                     |    Flags        |   ---- contains info such as no. of args
                                     -------------------
            FP                       |Comp. Fnct. Ptr.|    ---- points to beginning of compiled
    -----------------                -------------------        function
    |               |  ----------->  | Argument #0     |   ---:
    -----------------                -------------------      :
                                     | Argument #1     |      :
                                     -------------------      :-- passed from previous function
                                     |        :        |      :
                                     |        :        |      :
                                     -------------------      :
                                     | Argument #n     |   ---
                                     -------------------
                                     |        :        |   ---:
                                     |        :        |      :
                                     |        :        |      :-- current working area
            SP                       |        :        |      :
    -----------------                -------------------      :
    |               |  ----------->  | Top of Stack    |   ---
    -----------------                -------------------
                                     |                 |
```

Note that during normal processing, the value of the SP will be somewhat higher than the FP. Also, during the time that each particular function is being executed, the FP remains constant while the SP changes as the stack is pushed and popped.

COMPILED FUNCTIONS

As mentioned before, all macrocode is grouped into compiled functions. The compiled function block contains both macroinstructions and data. While a function is being executed, the location in the stack one below the address pointed to by the FP contains the the compiled function pointer. This is the virtual address of the first macroinstruction in the function, and has the data-type 'dtp-compiled-function.' In the 3600, THE COMPILED FUNCTION MACROCODE ALWAYS BEGINS WITH THE ODD-HALF INSTRUCTION.

This illustrated below:

```
                                                         Compiled Function in Virtual Memory
                                                              |                     |
                                                              -----------------------
                                                              |      Header         |
             ↓ Increasing Addresses ↓                         -----------------------
                                                              |         :           |
                                                              |         :           |
                                                              |         :           |
                                                              -----------------------
                                                              |       Data          |
                                  AMEM                        -----------------------
                         |                   |                |       Data          |
                         ---------------------                -----------------------
       FP                |Comp. Function Ptr| ------:          |       Data          |
  -----------------      ---------------------      :         -----------------------
  |               | ---> |                   |      -----> | 1st Inst |/////////|
  -----------------      ---------------------                -----------------------
                         |                   |                | 3rd Inst | 2nd Inst|
                                                              -----------------------
                                                              | 5th Inst | 4th Inst|
                                                              -----------------------
                                                              |         :           |
                                                              |         :           |
                                                              |         :      _____|
                                                              -----------------------
                                                              | Last Instructions   |
                                                              -----------------------
                                                              |                     |
                                                                    ↑          ↑
                                                                   ODD        EVEN
                                                             INSTRUCTIONS   INSTRUCTIONS

                                                           (Note that this is the reverse
                                                            of how instructions are displayed
                                                            by LCONS.)
```

MACROINSTRUCTION ADDRESSING MODES

All 3600 macroinstructions require one or more arguments. In some cases, the arguments will be the first one or several locations at the top of the stack. In other cases, the macroinstruction specifies an argument that resides somewhere else in the stack frame or in virtual memory. An argument can be either the source of data, as in the case of a PUSH instruction, or the destination for data, as in a POP. (Remember that we are talking about arguments to macroinstructions, as opposed to arguments to compiled functions.)

The following is a description of the more common ways that macroinstructions specify arguments. These are generally referred to as addressing modes:

1) Immediate (LCONS symbol: IMMED n, where n is a number). The argument is contained in the operand field of the macroinstruction.

2) Local (LOCAL SP|n or FP|n) The argument is an AMEM location, relative to either the SP or the FP. For example, FP|0 means that the argument is the location in AMEM pointed to by the FP; SP|2 means that the argument is two locations past the address pointed to by the SP. (Note, for instance, that FP|1 could be used to reference the second argument passed to a compiled function; FP|-3 references the location three addresses below the value in the FP, which normally contains the old PC.)

3) Constant (CONSTANT 'XX|n). The argument resides in the data portion of the compiled function. The value n is the displacement below the address of the location containing the first macroinstruction. That is, 'XX|0 references the location immediately prior to the first instruction, 'XX|3 is four locations before the first instruction.

4) Indirect (-INDIRECT @XX|n). This is simiiar to CONSTANT, except that the word at dispacement n before the first instruction contains the virtual address of the argument. This is illustrated below:

```
                      Addressing for POP-INDIRECT @XX|2


                                             Compiled Function
                                             in Virtual Memory

      ↓ Increasing Address ↓                 |               |
                                             -----------------
                                             |               |
                                             -----------------        Virtual Memory
                                  XX|2 -->|     Address      | ----:
                                             -----------------      :  |                 |
                                             |               |      :  -------------------
                                             -----------------      ----> |   Argument   |
                            AMEM             |               |         -------------------
                                             -----------------         |                 |
                      |               |   :-> | 1st Ins|///////|
                      -----------------   :  -----------------        The above location
       FP             |Comp. Fcn. Ptr.| ---  | 3rd Ins|2nd Ins|       receives the data
  ------------        -----------------      -----------------        popped off the stack.
  |          | --->   |               |      |               |
  ------------        -----------------
                      |               |
```

To give an example of indirect addressing, suppose that the current macroinstruction is PUSH-INDIRECT @XX|2, and that the FP contains 17777774006. The previous location in AMEM, i.e., 4005@A, contains 14000034160, which is the compiled function pointer. The displacement XX|2 means that we need to look at the location three addresses below the value of the compiled function pointer, i.e., 34155@V. This location contains 1200005616 (dtp-locative), which is the address of the argument we're interested in. Location 5616@V contains 4000364110 (dtp-fix). This the value that gets pushed on the stack.

NOTE

The address displacements (XX|n, SP|n, FP|n) displayed by LCONS are in DECIMAL rather than octal. The author makes no claim to knowing why LCONS does it this way, other than to provide a trap for young players.

MORE ABOUT MACROINSTRUCTIONS

The following are some comments intended to clarify the functions of certain macroinstructions, and their symbolic representation under LCONS. This is obviously not a complete list, but hopefully will be expanded over time.

1) BRANCH
   BRANCH-TRUE
   BRANCH-FALSE

These are the basic-decision making macroinstructions. The conditional instructions BRANCH-TRUE and BRANCH-FALSE look at the top word of the stack. If this word is the symbol 'nil' (201100000), the condition is considered false -- BRANCH-FALSE will branch, and BRANCH-TRUE will not, but instead will go onto the next sequential macroinstruction. Likewise, if the top of the stack contains anything but the 'nil' symbol, BRANCH-TRUE will branch and BRANCH-FALSE will not. The nonconditional BRANCH instruction branches no matter what.

But where does the program branch to? All branch addresses are PC-relative, i.e., they specify a number to be added to or subtracted from the PC, rather than an absolute branch address. LCONS represents the branch displacement in the following manner:

PC|nw * xh

Where n is the number of words to be added or subtracted from the PC, and x specifiies which half. Specifically, x = 0 means 'same half' as the current PC, and x = 1 specifies 'other half.' For example:

a) PC = 140000027222
   macroinstruction = BRANCH PC|3w*0h
   program branches to: 27225, even half (new PC = 140000027225).

b) PC = 160000014732
   macroinstruction = BRANCH-FALSE PC|-5w*1h
   top of stack = 201100000 (dtp-nil)
   program branches to: 14725, even half (new PC = 140000014725)

The hardware mechanism for this is worth mentioning. The branch displacement is the 8-bit operand field in the macroinstruction. Bit 0 specifies the 'half' displacement, bits 1-7 are the 'word' displacement, with bit 7 the sign bit. This allows branch displacements of up to plus-or-minus 77 octal words. When the branch is executed, the operand field is gated onto the BBUS and rotated left 31 bits, so the 'half' bit is now in the bit-31 position, and the 'words' bits are in 0-6. This is added to the PC. Since the carry-out from ALU bit 31 goes in the 'bit-bucket,' having the 'half' bit set will, in effect, complement PC bit 31. The result is loaded back into the PC, and the next instruction is fetched from this address.

VIRTUAL ADDRESSING

The 3600 uses a virtual memory addressing scheme, which permits the writing of programs and data structures that are much larger than the physical main memory. This is accomplished by storing most of the 'virtual address space' on disk and keeping only the part that is currently being accessed in main memory. As the macrocode branches to different areas in virtual memory, the system automatically writes the no-longer-needed areas back onto disk, and reads the new active areas into main memory.

The virtual addressing system is implemented by a combination of hardware, microcode, and macrocode. This discussion will focus on the hardware and microcode parts, since these are the most relevant to hardware troubleshooting, and make one grand hand-wave over the macrocode part. For additional information refer to:

```
        >lmach>storage.text
        >lmach>ucode>map.lisp
```

It is important to understand the difference between virtual and physical memory. Virtual memory is what the macrocode references, by means of 28-bit virtual addresses. The PC, compiled function pointers, and constant and indirect adddresses are all virtual addresses. Physical memory consists of the LBUS main memory and AMEM. The physical address is 24 bits long, although a great deal of the physical address space has no memory actually installed. Bits 19 through 23 of the physical address select the LBUS chassis slot, while 0 through 18 select a word location within the board in that slot. The DP A-memory (AMEM) resides in the very top 4k of physical address space.

The organization of virtual and physical address spaces are shown below:

```
                        VIRTUAL ADDRESS SPACE

               -----     --------------------  -----
                 ↑       |                  |    ↑
                 |       |                  | Addresses
                 |       |                  |1700000000 - 1777777777
                 |       |                  |map directly into
                 |       |                  | physical memory
                 |       |                  |    ↓
                 |       |------------------|  -----
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                         |                  |
              28 Address |                  |
                Bits     |                  |
              256M Words |                  |
                         |                  |               (NOT TO SCALE)
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
                 |       |                  |
   |      |                   |
   |      |                   |
   |      |                   |
   |      |                   |
   |      |                   |
   |      |                   |
   |      |...................| <-- %WIRED-VIRTUAL-MEMORY-HIGH
   |      |                   |   ( 36377 per Rev. 214)
   ↓      |                   |
 -----    ---------------------
```

VIRTUAL ADDRESSING (cont'd)

```
                   PHYSICAL ADDRESS SPACE
                                               ↓
     -----        -------------------     ---
       ↑          |                 | Addresses 77770000 - 77777777 are in AMEM
       |          |-----------------|     ---
       |          |                 |      ↑
       |          |                 |
       |          |                 |
       |          |                 |
       |          |                 |
       |          |                 |
       |          |                 |
       |          |                 |
       |          |                 |
       |          |                 |
  24 Address Bits |                 |          (NOT TO SCALE)
     16M Words    |                 |

       |          |      etc.       |
       |          |                 |
       |          |                 |
       |          |                 |
       |          |-----------------|  <-- 04000000
       |          |                 |
       |          |                 |
       |          |     SLOT 1      |
       |          |                 |
       |          |                 |
       |          |                 |
       |          |-----------------|  <-- 02000000
       |          |                 |
       |          |                 |
       |          |                 |
       |          |     SLOT 0      |
       |          |.................|  <-- %WIRED-PHYSICAL-ADDRESS-HIGH (36377 per Rev.
       |          |                 |                                           214)
       ↓          |                 |
     -----        -------------------  <-- %WIRED-PHYSICAL-ADDRESS-LOW (0 per Rev. 214)
```

Whenever the macrocode needs to access a virtual address (either as data or macroinstructions) the system must translate or 'map' the virtual address into a physical address. The mapping process is dynamic, since the contents of physical memory changes as areas are swapped to and from disk. Mapping is done on a 256-word page basis. The virtual address can be thought of as a 20-bit 'virtual page number' (VPN) and an 8-bit word offset:

```
                  VIRTUAL ADDRESS

      27                          8  7        0
      ------------------------------------------
     |               VPN          | Word Offset |
      ------------------------------------------
```

The VPN is mapped into a 'physical page number' (PPN), which together with the word-offset, forms the physical address:

```
                    PHYSICAL ADDRESS

           23                    8  7         0
           ---------------------------------------
          |          PPN          | Word Offset |
           ---------------------------------------
```

(The physical word offset is always equal to the virtual word offset.)

VIRTUAL ADDRESSING (cont'd)

The mechanism for mapping is based on a series of tables stored in various memories. In principle, the system could use one large table, which would have one entry for every VPN, and each entry would contain either a PPN or a disk address, depending on whether that page was currently in physical memory. However, this would require a minimum of 1M words of memory, and would impose an extra LBUS cycle for every virtual memory reference. In practice, the 3600 uses a series of tables which contain an increasing number of entries, and likewise, are increasingly slower to access. In other words, the system looks in the smallest, fastest table first, and if it doesn't find an entry for that VPN, goes on to the next larger, slower table, and so on.

The two tables of interest here are the two fastest, the 8k-entry Map Cache located on the MC board, and the Page Hash Table Cache (PHTC), which is always stored in a specific area in main memory. Generally speaking, each entry in both of these tables contains a 'tag', which is used to uniquely identify that entry with a VPN, and a PPN, which the VPN will map to. (Note that the Map Cache is organized as two 4K halves, called 'A' and 'B' halves or sectors.)

The mapping procedure works like this, in this order:

1) The microcode loads the virtual address into the Virtual Memory Address (VMA) register on the MC board. A subsequent microinstruction starts an LBUS read or write cycle.

2) The MC hardware checks to see if VMA bits 24-27 are all set to 1's. If so, it means that the VMA is mapped directly to a physical address. VMA bits 0-23 are gated onto the LBUS address lines, and the memory cycle completes.

3) If VMA 24-27 do not all equal 1, the VPN bits of the VMA are 'hashed' by means of exclusive-or gates, and the hashed address references an entry in the Map Cache. (Both the 'A' and 'B' halves of the Map are read simultaneously.) The tag portion of both Map entries is compared with the VPN. If a tag matches, it is called a 'map hit,' and the PPN from the Map is gated onto LBUS address lines 8-23. The word offset, i.e., bits 0-7 come directly from the VMA. (NOTE -- the hardware is designed such that the MC should never get a hit in both the 'A' and 'B' halves of the Map at the same time -- this is an error condition. Also note that the address-hash and tags use some bits called ASN 0-7. These are the 'Address Space Number,' and come from another MC register. The ASN permits use of multiple virtual address spaces, but this is not currently implemented in 3600 software.)

4) If the VPN does not match either Map tag, a condition called a 'map miss' exists. The MC causes a microcode trap to CMEM address 10001, which is the start of the MAP-MISS routine. At the same time, the MC generates a PHTC Probe Address, PHTA 0-23. This is based on a combination of bits from a base register and hashed VMA bits, and is the address of an entry in the PHTC table in main memory. The PHTC Probe Address is gated onto the LBUS, and the MC initiates a read cycle. When the data from memory becomes valid, it is written into the least-recently-used half of the Map at the same hashed-VMA address.

5) The microcode checks for a tag match between the VPN and the new entry read from the PHTC. If they match, the microcode restarts the original LBUS cycle, taking the PPN from the new Map entry, and branches out of the trap.

6) If a PHTC-miss occurs, the microcode tests for two conditons:

a) The VMA is within a area called thê stack buffer, as defined by the contents of AMEM locations %STACK-BUFFER-LOW and %STACK-BUFFER-LIMIT. If so, it means the VMA maps into the stack area in AMEM.

b) The VMA is equal to or less than an address contained in AMEM location %WIRED-VIRTUAL-ADDRES-HIGH. This is a part of virtual memory that is, by definition, always mapped into a specific area in physical memory (defined by %WIRED-PHYSICAL-ADDRESS-LOW and %WIRED-PHYSICAL-ADDRESS-HIGH), and is never swapped out.

If one of the aove conditons is met, the microcode generates the appropriate tag and

PPN, and writes it into the least-recently-used Map half. The LBUS cycle is restarted, and the microcode branches out of the MAP-MISS trap.

7) If none of the above conditions can be met, it is called a 'page-fault.' The microcode calls a macrocode funtion, pointed to by the contents of AMEM location FUNCTION-PAGE-FAULT. The macrocode searches through some larger tables for the VPN, and if necessary, swaps the page in from disk.

NOTE

Neither a map-miss nor a page-fault is an error condition, although a second page-fault during the page-fualt handling function can be an error.

600 Architecture Documentation

[There is an invisible disclaimer here that says that your fingers will be
cut off if you use this information as the basis for manufacturing equipment,
or words to that effect. I'm sure Clark will put one here before too long.]

Table of Contents

> Introduction

All symbols in this document are accessible from the SYS package.

[Describe the different ways of naming fields: some are byte fields,
most are named by their accessor functions/substs/macros.]

> Object Representation

>> Object References

The fundamental data object manipulated by Lisp is an Object Reference. These are sometimes called Pointers or Qs (long ago in the old days "Q" was an abbreviation for "Quad Byte", i.e. 32-bit word, I suspect). The value of a variable, an argument to a function, or an element of a (general) array is an object reference.

An object reference is represented as a 34-bit quantity consisting of a data type code and either a 28-bit address or an immediate quantity of 28 or 32 bits (depending on the data type).

>> Memory Words

A word in memory consists of 36 bits. 34 of these bits contain either an object reference or a special memory word consisting of a data type code that is not used in object references and a 28-bit address or immediate field. Such special memory words are used for object headers, invisible pointers, unallocated memory, and unbound-variable markers; see below for descriptions of these.

The other 2 bits of a word in memory are called the cdr code; their most common use is in compact representation of lists. In some cases where the contents of a memory is not an object reference, the cdr-code bits are used as an extension of the rest of the word. Examples of this include headers and macrocode instructions.

When a memory word contains "just bits", rather than an object reference, the convention is to store a fixnum in the word and use the 32 bits of the fixnum as the bits. This ensures that all memory words contain a legal data-type code. Examples of the use of this include compiled functions and arrays of packed bytes.

>> Object Representations

A lisp object is usually represented as some memory words; the address of one of these memory words is placed in the address field of an object reference to this object. There are three fundamentally different classes of object representation: list, structure, and immediate.

A list object is a cons or an object built out of conses. The representation consists of a block of memory words strung together by means of the cdr codes. Often the block consists of only one or two words, so it is important to avoid the overhead of having an extra header word; this is why list representation and structure representation are different. Currently conses, lists, and closures have list representations.

A structure object is represented as a block of memory words. The first word contains a header with a special data type code. Usually all words after the first contain object references. The header contains enough information to determine the size of the object's representation in memory. Furthermore it contains enough information about the type of the object so that a legal object reference designating this object can be constructed. Structure representation is designed to work for large objects. Some attention is also paid to minimizing overhead for small objects, but there is always at least one word of overhead. Most objects are represented as structures.

An immediate object is not represented with any memory words. Instead the entire object representation is contained right in the object reference. To be an immediate object, an object type must not be subject to meaningful side-effects, must have a small representation, and must have a need for very efficient allocation of new objects of that type. Currently small integers (fixnums), single-precision floating-point numbers, and characters have immediate representations.

The question "When an element in an art-q array is set to a fixnum or to a single-precision flonum, are these bits copied into it, or is an object reference made?" is not meaningful. Because fixnums and single-precision flonums are immediate objects, there is no difference between the two alternatives.

The term "object reference" implies two entities, one referring to the other. At the level of concrete words and bits, list and structure objects fit this model, but immediate objects do not. An immediate object reference refers to an abstract object that has no physical representation outside of the object reference itself.

The 3600 never uses the "unboxed" objects used by some other Lisps, such as the LM-2, Maclisp on the pdp10, and Interlisp on the pdp10.

>> Pointers

There are two special kinds of object references that refer to a particular memory cell in the representation of an object, rather than to an object as a whole. These "pointer" object references are mainly for system programming.

A locative pointer contains the address of a memory word. The data type code is dtp-locative. In addition to explicit use of locatives, the representation of an intermediate state of a computation often contains memory addresses represented as locatives.
Locatives are described in the manual.

A PC contains the address of an instruction in a compiled function. This address consists of the address of a word containing a pair of instructions, and a designation of which halfword is addressed. Two data type codes are used; one for the even halfword (dtp-even-pc) and one for the odd halfword (dtp-odd-pc).

>> Storage Conventions

These conventions ensure that low-level memory operations always preserve the high-level object model and guarantee that the incremental garbage collector always sees consistent structures in memory. Objects that have live object references designating them do not have their memory reclaimed, uninitialized memory is not referenced, and memory that represents unreferenced objects is eventually reclaimed. In addition, these conventions ensure that the %FIND-STRUCTURE-HEADER, %FIND-STRUCTURE-LEADER, and %STRUCTURE-TOTAL-SIZE subprimitives can work.

Given an object reference, complete information about the object including the size of the block of memory used to represent it can be computed. Some of this comes from the data-type code in the object references, and the rest comes from memory, mainly from the header word (if structure representation is used).

Given just the representation in memory of an object, the same information has to be discoverable. This can happen in two ways. When the garbage collector's scavenger is operating, it scans successively through the memory representations of objects that are known (or assumed) not to be garbage. The scavenger pays no attention to the boundaries between objects' representations; it is only interested in finding all the embedded object references. When the garbage collector's transporter is operating, it is given a reference to an object and must copy the object's representation from one part of memory to another (from oldspace to copyspace). The reference could be a locative pointer into the middle of an object, so it is necessary to find the whole containing structure, and discover its size
and type, when given a pointer into any legal place
in the middle.

This generates rules for how objects and pointers can be legally used:

>>> Numbers vs. locatives

Sometimes addresses are represented as numbers (fixnums) and sometimes they are represented as locative pointers. How does one decide which to use? The difference between them is that the garbage collector recognizes a locative as an address, but does not recognize a number as an address. By using a locative pointer, one ensures that the object containing the memory word being addressed will be retained and will not be discarded as garbage. Furthermore, if the garbage collector moves the object's representation to a different address the locative pointer will be relocated. Locative pointers should always be used except when a number is required for specific low-level reasons. When it is to deal with virtual memory at a lower level than the Lisp object abstraction, addresses should be represented as numbers. This ensures that the garbage collector will not recognize them as addresses, which makes it possible to deal with addresses that would otherwise be illegal, e.g. addresses of memory locations that have not been allocated to objects.

In the 3600, fixnums have 32 bits but addresses have only 28 bits. This makes it unnecessary to worry about overflow when dealing with addresses as numbers, and guarantees that the address field of any number that represents an address will contain the address (this would not be the case if the number overflowed into a bignum; the address field would contain the address of the representation of the bignum, not the actual value of the bignum). This guarantee means that numbers representing addresses may freely be used as arguments to such subprimitives as %MAKE-POINTER, %P-DPB, and %BLOCK-STORE-TAG-AND-POINTER.

Where is it legal for pointers to point? This refers to locative pointers, to the addresses in object representations, and to addresses in special memory words such as forwarding pointers. It is illegal to have an address that points to the normal part of virtual memory, but does not point to an object. It is illegal to construct new pointers to oldspace. In order to be sure that the garbage collector can find all pointers to objects in oldspace, evacuate the objects, and relocate the pointers, it is necessary that the only operation that can create pointers to oldspace be the flip operation that makes some space into oldspace.

This means that it is illegal to develop a pointer that points to the next word after the representation of an object, for use as an exclusive upper bound. That next word might be in unallocated storage that is not yet part of an object. Worse, it might be in a different space if the space containing the object was completely full; hence it might be in oldspace.

It is illegal to point to locations in the stack above the top of the stack, since the contents of those locations is undefined and is not protected by the garbage collector. This is why the stack-pointer points to the last valid location in the stack rather than the first invalid location. The frame-pointer can momentarily point one location past the end of the stack, when a frame is empty, but this is only momentary and can never be true in any stack group other than the one that is currently in active execution.

It is legal to point to any location that is not part of the normal virtual memory, i.e. A-memory locations, physical-address locations, and communication-area locations. (See the Address-space organization section.)

--- probably there is more to say here? And did I get all the invariants
--- we are trying to guarantee?

All memory locations contain a data-type tag that at least says whether or not there is an address in that memory location. There are no "unboxed Qs", to use LM-2 terminology. It is illegal to use all 36 bits of a memory location to store data. You can actually cram 35 bits of data into a memory word by ensuring that the data type tag is either dtp-fix or dtp-float, which fits one bit of information into two bits of memory, and then storing arbitrary information in the other 34 bits.

By being excruciatingly careful in what subprimitives you use, so that the GC tag bits never get turned on, you actually can use all 36 bits. The cold-load generator does this, unfortunately. It does not have the seal of approval.

The data type codes that denote neither genuine object references nor pointers, i.e. the special-word codes, must be used in a controlled fashion. A memory word containing a header data type cannot be passed around as if it were an object reference, for if the garbage collector, or anyone else, looked at memory while that word was being passed around it would see something that it would take to be a legitimate object header, in the middle of another object or in the middle of a stack, and would become quite confused. One may not generate a header in a variable and then store it. One may not even pass it as an argument; it has to be generated in the place where it is finally going to be, using, for example, %P-DPB, %P-DPB-OFFSET, or %P-STORE-TAG-AND-POINTER.

> Details of object representations

The information in this section is available in machine-readable form in the file SYS: L-SYS; SYSDEF LISP.

>> Fixnum dtp-fix

An immediate object consisting of a 32-bit two's-complement integer.

>> Single-float dtp-float

An immediate object consisting of a 32-bit IEEE single basic floating-point number. The following fields are defined:

%%SINGLE-SIGN	0 for positive numbers, 1 for negative numbers
%%SINGLE-EXPONENT	excess-127. exponent
%%SINGLE-FRACTION	positive fraction, with hidden 1 on the left

>> Character dtp-character

An immediate object containing the following fields in its 28 bits of immediate data.

%%CHAR-BITS	Control, Meta, Super, Hyper bits
%%CHAR-STYLE	Italic, large, bold, etc.
%%CHAR-CHAR-SET	Character set
%%CHAR-SUBINDEX	Index within this character set

The numbers that go in the style field are allocated dynamically as required and are only meaningful with respect to a particular character set. Zero always means the default style. Fonts of different size or appearance that logically represent the same characters are considered styles.

The numbers that go in the character set field are allocated dynamically as required, except for zero which always means the standard character set (Lisp-machine extended ascii). Characters in different character sets are never considered equal. Some character sets, such as Japanese Kanji, contain more than 256 characters and therefore use multiple character set numbers.

The following fields are combinations of the above primitive fields:

%%CHAR-DEVICE-FONT	Style concatenated with character set. Devices such as the screen and the LGP use this field to select a "font" of characters.
%%CHAR-ALL-BUT-SUBINDEX	All bits but the subindex.
%%CHAR-CODE	Character set and character therein.

>> Conses and lists dtp-list

The address field contains the address of a list representation. Let A be the address in the pointer, C(A) be the contents of the memory word whose address is A, and CC(A) be the cdr code of that memory word. C(A) is an object reference; CC(A) is a 2-bit number.

The car of the cons or list is C(A). The cdr is determined by CC(A). The following cdr codes are defined:

cdr-normal	the cdr is C(A+1)
cdr-next	the cdr is a list whose address is A+1
cdr-nil	the cdr is nil

The fourth possible value of CC(A) is not currently used.

A list representation consists of a contiguous block of one or more memory words. The cdr code of the last word is always cdr-nil. The cdr code of the second-to-last word may be cdr-normal or cdr-next. The cdr code of every other word is cdr-next. Note that when cdr-normal is used to represent a 2-word cons the cdr code of the second word is always cdr-nil.

Note that a dtp-list pointer can point into the middle of a list representation. This happens any time cdr-next is used; for instance, if a list of four elements is fully cdr-coded, its representation consists of four words. The contents of each word is an element of the list. The cdr codes of the first three words are cdr-next; the cdr code of the last word is cdr-nil. An object reference to the CDDR of this list has data type dtp-list and the address of the third word. The garbage collector protects the entire block of storage if any word in it is referenced.

The first word in a list representation is found by searching backwards for a word that is preceded by a cdr-nil word. (This is oversimplified; there are several boundary conditions and special cases in reality.)

The rplacd operation interacts with cdr coding. Rplacd of a cons represented with cdr-normal simply stores into the second word. But rplacd of a cons represented with cdr-next or cdr-nil must change the representation so that the cdr is represented explicitly before it can be changed. There is one exception; if the cdr is being changed to NIL, the cdr-nil cdr code is used to represent it. This can split an object representation into two independent object representations, one of which might then be garbage-collected. dtp-header-forward is used for this; see below.

>> Structure Headers dtp-header-p or dtp-header-i

The first word in a structure representation is a header word. It always contains one of the data type codes dtp-header-p, dtp-header-i, or dtp-header-forward. This special data type marks the boundary between object representations in memory. dtp-header-forward is used when the object's representation has been moved elsewhere in memory, typically so that it can be made larger. See below. The 28-bit immediate field of dtp-header-p contains an address, whereas the 28-bit immediate field of dtp-header-i simply contains bits. dtp-header-p can be thought of as being equivalent to a locative-pointer object reference.

The %%HEADER-TYPE-FIELD of the header word (another name for the cdr code) identifies the particular type of object. The following values for this field are defined:

```
    For dtp-header-p headers:
        %HEADER-TYPE-SYMBOL              a symbol
        %HEADER-TYPE-INSTANCE            an instance of a flavor
    For dtp-header-i headers:
        %HEADER-TYPE-COMPILED-FUNCTION   a compiled function
        %HEADER-TYPE-ARRAY               an array
        %HEADER-TYPE-NUMBER              an extended number
```

Extended numbers have a %%HEADER-SUBTYPE-FIELD which identifies the particular type of number. Possible values are:

```
        %HEADER-TYPE-BIGNUM
        %HEADER-TYPE-RATIONAL
        %HEADER-TYPE-COMPLEX
        %HEADER-TYPE-DOUBLE
```

>> Symbols dtp-symbol or dtp-nil

An object reference to the distinguished symbol NIL uses data type code DTP-NIL so that it can be more easily recognized, especially by the CAR and CDR operations.

A symbol is represented as a 5-word structure. The header has data type DTP-HEADER-P and header type %HEADER-TYPE-SYMBOL. The address field of the header contains the address of the symbol's print-name, a string. A string is a type of array. The words in a symbol's representation are:

SYMBOL-PNAME-CELL	the header
SYMBOL-VALUE-CELL	contains the value or an unbound marker
SYMBOL-FUNCTION-CELL	contains the definition or an unbound marker
SYMBOL-PROPERTY-CELL	contains the property list
SYMBOL-PACKAGE-CELL	contains the home package, or NIL

An unbound marker has data type DTP-NULL and the address of the symbol in its address field. This is not an object reference; it is a special memory word. Reading an unbound marker from any memory location causes an error, except when it is read by a suitable subprimitive; the BOUNDP primitive effectively uses the %P-DATA-TYPE subprimitive internally to check for an unbound marker.

>> Bignums dtp-extended-number

The header word contains data type DTP-HEADER-I, header type

%HEADER-TYPE-NUMBER, and header subtype %HEADER-TYPE-BIGNUM.
The following fields in the header word are specific to bignums:

%%BIGNUM-SIGN	0 for a positive number, 1 for a negative number
%%BIGNUM-LENGTH	the number of "digits" that follow

Following the header is a sequence of "digits" representing the magnitude of the bignum. The least-significant digit is stored first. Each digit is a positive fixnum containing 31 bits of the value of the bignum.

[There are thoughts of changing the format of bignums to be two's complement and to use all 32 bits for each digit. This bignum sign bit is the value of all the most significant bits not explicity stored in the bignum. Therefore, -1_32. would occupy 2 words: the header with sign 1 and length 1, and a fixnum of 0. 1_31. would also occupy 2 words: the header with sign 0 and length 1, and a fixnum that happens to be -1_31.]

>> Ratios dtp-extended-number

The header word contains data type DTP-HEADER-I, header type %HEADER-TYPE-NUMBER, and header subtype %HEADER-TYPE-RATIONAL. Two words follow, containing object references for the numerator and the denominator. These fields are named SI:RATIONAL-NUMERATOR and SI:RATIONAL-DENOMINATOR.

>> Double-floats dtp-extended-number

The header word contains data type DTP-HEADER-I, header type %HEADER-TYPE-NUMBER, and header subtype %HEADER-TYPE-DOUBLE. Following are two fixnums, containing the sign, exponent, and fraction as packed fields. The most-significant word is stored first, violating normal byte-order conventions. The second fixnum contains the low 32 bits of the fraction. The first fixnum contains the following fields:

%%DOUBLE-SIGN	0 for a positive number, 1 for a negative number
%%DOUBLE-EXPONENT	Excess-1023. exponent
%%DOUBLE-FRACTION-HIGH	Top 20 bits of fraction (excluding the hidden bit)

In non-generic code double-precision floating-point numbers are often represented with a special immediate representation as a pair of fixnums. Avoiding the normal in-memory object representation saves consing overhead.

>> Complex numbers dtp-extended-number

The header word contains data type DTP-HEADER-I, header type %HEADER-TYPE-NUMBER, and header subtype %HEADER-TYPE-COMPLEX. Following are presumed to be two object references for the real and imaginary parts. However, complex numbers are not actually implemented.

>> Instances dtp-instance

The header contains data type DTP-HEADER-P, header type %HEADER-TYPE-INSTANCE, and the address of the flavor description structure. This structure is also found on the FLAVOR property of the name of the flavor, and contains fields defined by the defstruct named **FLAVOR**.

Words after the header are value cells of instance variables. Each word contains either an object reference or an unbound marker. The cdr codes are not used.

The architecture (as opposed to the flavor system proper) only specifies a few of the fields in the flavor description structure. Other types of instances based on other message-receiving paradigms could be implemented. Some non-microcode parts of the system do currently assume that all instances are instances of flavors.
The following are the architectural fields in a flavor description structure. The flavor system has its own names for these.

%INSTANCE-DESCRIPTOR-HEADER	space reserved for an array header
%INSTANCE-DESCRIPTOR-RESERVED	space reserved for a named-structure symbol
%INSTANCE-DESCRIPTOR-SIZE	number of words in an instance (one less

	than the number of instance-variable slots)
%INSTANCE-DESCRIPTOR-BINDINGS	for special instance variables; see below
%INSTANCE-DESCRIPTOR-FUNCTION	message handler hash table
%INSTANCE-DESCRIPTOR-TYPENAME	a symbol which is returned by TYPEP

There is a feature that if %INSTANCE-DESCRIPTOR-FUNCTION is not an array, then it is a function to be called. However this feature is not currently implemented by the microcode.

%INSTANCE-DESCRIPTOR-BINDINGS can be NIL, the normal case, or a list. The list must be cdr-coded; the microcode assumes this for a little extra speed. Each element of the list is either a locative pointer to a symbol's value cell or a fixnum. The symbol value cells are bound to external-value-cell-pointers to successive instance variables, starting with the first instance variable. A fixnum is a count of the number of instance variables to be skipped over without binding any value cell to them.

>> Arrays dtp-array

The header word contains data type DTP-HEADER-I, header type %HEADER-TYPE-ARRAY, and a number of bits and byte fields that define the particular characteristics of the array. There are a large variety of array formats, designed so that small arrays that don't use all the possible features have less overhead in both space and time.

An array's representation consists of three parts: the prefix, the leader, and the data, stored in memory in that order.

The prefix consists of the header and optionally some additional words. The prefix defines the type of array and the type of its elements, the dimensions of the array, the sizes of the leader and data, and whether the array is a named structure. The prefix includes all the information needed to access the other parts of the array as well as all the information needed to determine the size of the array's representation in memory. Details of the prefix for each type of array are given below.

The leader is an optional 1-dimensional array of object references which the user can use to store random Lisp objects associated with but not part of the array. Array leaders are described in the manual.

The data portion of an array represents its elements. Indirect and displaced arrays have no data of their own. For ART-Q, ART-Q-LIST, and ART-CHAR-Q-STRING arrays, the data are a block of object references. The other array types have elements that are numbers of one sort or another; the data are a block of fixnums and the elements are packed into the 32-bit words of the fixnums, as many per word as will fit. The TV buffer is compatible with this so that bit/byte arrays can be displaced into it. ART-Q-LIST arrays use the cdr codes to embed a list of its elements in the array. All other array types do not use the cdr codes.

The prefixes of each type of array currently implemented are as follows. These fields exist in all types of arrays, and hence are in the first word of the prefix, which is the header word:

ARRAY-NAMED-STRUCTURE-BIT	1 for named structures, 0 normally
ARRAY-SPARE-FLAG-BIT	a bit that is not used
ARRAY-DISPATCH-FIELD	see below
ARRAY-TYPE-FIELD	see below

The ARRAY-TYPE-FIELD contains a value that defines the high-level or user-visible type of the array. The values of the constants ART-1B, ART-2B, ART-Q, etc. are the values that appear in this field.

The ARRAY-DISPATCH-FIELD defines the particular form of the prefix and selects a particular access method for elements of the array. The array microcode starts by dispatching on the contents of this field. The remainder of this section describes each possible value for this field and what it implies about the rest of the array's prefix, its leader, and its data.

>>> Simple arrays

%ARRAY-DISPATCH-1-BIT
%ARRAY-DISPATCH-2-BIT
%ARRAY-DISPATCH-4-BIT
%ARRAY-DISPATCH-8-BIT

%ARRAY-DISPATCH-16-BIT
%ARRAY-DISPATCH-WORD
%ARRAY-DISPATCH-CHARACTER
%ARRAY-DISPATCH-BOOLEAN

These dispatch codes define simple one-dimensional arrays with no leader and no indirection or displacement. The particular dispatch code specifies the type of data. The first five codes specify packed bytes of various sizes; %ARRAY-DISPATCH-WORD specifies object references; %ARRAY-DISPATCH-CHARACTER specifies character objects with zero in all fields except %%CHAR-SUBINDEX, packed four per word; %ARRAY-DISPATCH-BOOLEAN specifies array elements that are either T or NIL, represented as bits packed 32 per word.

The prefix is one word long. ARRAY-NORMAL-LENGTH-FIELD contains the number of elements in the array.

>>> One-dimensional arrays with leaders

%ARRAY-DISPATCH-LEADER

This dispatch code defines an one-dimensional array with a leader, but no indirection or displacement. The ARRAY-TYPE-FIELD is consulted to determine the type of the elements. The prefix is one word long and contains:

ARRAY-SHORT-LENGTH-FIELD	The number of elements in the array
ARRAY-LEADER-LENGTH-FIELD	The number of elements in the leader

If the leader exists and its first element is an integer, then it is the fill-pointer.

>>> Simple indirect arrays

%ARRAY-DISPATCH-SHORT-INDIRECT

This dispatch code defines a one-dimensional indirect or displaced array without a leader. The ARRAY-TYPE-FIELD is consulted to determine the type of the elements. The representation of the array consists solely of the prefix. The prefix is two words long and contains the following fields:

ARRAY-SHORT-INDIRECT-LENGTH-FIELD	The number of elements in the array
ARRAY-SHORT-INDIRECT-OFFSET-FIELD	The index offset (0 if displaced)
ARRAY-INDIRECT-POINTER	Where the data are

The indirect-pointer is an object reference to an array if this array is indirect. If this array is displaced, the indirect-pointer is the address of the data, expressed as either a fixnum or a locative. A locative should be used if the data lie in normal virtual memory, inside an object. A fixnum is used when the data lie in special memory, such as in a TV buffer, which is addressed by physical address and cannot be moved by the garbage collector. A locative would work for this, too.

>>> General form for one-dimensional arrays

%ARRAY-DISPATCH-LONG

This dispatch code is used for a one-dimensional array that cannot use any of the above forms, because its data length, leader length, or index offset is too large to fit in the packed fields used by the above forms, or because these parameters need to be arbitrarily adjustable. The ARRAY-TYPE-FIELD is consulted to determine the type of the elements. The prefix is three or four words long. The following fields exist in the prefix:

ARRAY-LONG-PREFIX-LENGTH-FIELD	The number of words in the prefix
ARRAY-LONG-LEADER-LENGTH-FIELD	The number of elements in the leader
ARRAY-DIMENSIONS-FIELD	The number of dimensions (always 1)
ARRAY-INDIRECT-POINTER	Where the data are
ARRAY-LONG-LENGTH-FIELD	The number of elements in the array
ARRAY-INDEX-OFFSET-FIELD	The index offset

The first three fields are packed bytes. The last three fields are full words. The last field is only present in indirect arrays.

The indirect-pointer is a locative pointer to the location immediately after the leader if the array is neither indirect nor displaced. If the array is indirect, it is an object reference to an array. If this array is displaced, the indirect-pointer is the address of the data, expressed as either a fixnum or a locative. A locative should be used if the data lie in normal virtual memory, inside an object. A fixnum is used when the data lie in special memory, such as in a TV buffer, which is addressed by physical address and cannot be moved by the garbage collector. A locative would work for this, too.

>>> Simple two-dimensional arrays

%ARRAY-DISPATCH-SHORT-2D

This dispatch code defines a two-dimensional array with no leader, indirection, or displacement. The prefix is one word long and contains the following fields:

ARRAY-ROWS-FIELD	The first dimension
ARRAY-COLUMNS-FIELD	The second dimension

The number of elements in the data is the product of these two fields. Arrays are stored in column-major order, so the linear index into the data is the sum of the first subscript and the product of the second subscript times the number of rows.

>>> General form for multi-dimensional arrays

%ARRAY-DISPATCH-LONG-MULTIDIMENSIONAL

This dispatch code defines an array with more than one dimension. (Zero-dimensional arrays are not currently implemented). The fields in the prefix are the same as for %ARRAY-DISPATCH-LONG, described above, however some additional fields are appended, and of course the ARRAY-DIMENSIONS-FIELD is not 1. After the first 3 or 4 words of the prefix the dimensions of the array are stored as a sequence of fixnums. The contents of ARRAY-LONG-LENGTH-FIELD is the product of these dimensions. Following the dimensions are the subscript multipliers, which must be the last words in the prefix. The number of subscript multipliers is one less than the number of dimensions. Arrays are stored in column-major order, so the first subscript's multiplier is understood always to be 1. The linear index into the array is the sum of each subscript times its corresponding multiplier.

In a conformal indirect array, there is no direct relation between the subscript multipliers and the dimensions. The dimensions are whatever dimensions the user requested, but the subscript multipliers are based on the array that is indirected into.

When we switch to row-major order for arrays, the representation of multi-dimensional arrays will be changed.

>> Stack groups dtp-array

A stack group is simply a named structure, which is a kind of array. The fields in this structure define several stacks and the state of a computation executing on them. See the section on stacks for details.

>> Compiled Functions dtp-compiled-function

Compiled functions are exceptional in that the object reference to a compiled function does not contain the address of the first memory word in the representation. Instead it contains the address of the first instruction (called the entry instruction). This speeds up function calling since the entry instruction is the first thing in the compiled function that must be accessed in order to call it, and the header is not needed at all when calling a compiled function.

The representation of a compiled function consists of three major parts, stored in memory in this order: The prefix is four words, the first of which is the header; the fields in the prefix are described below. The external reference table follows; it contains constants, locative pointers to function-definition cells, and locative pointers to special-variable value cells. The macroinstructions follow; the 17-bit instructions are packed two per word. Each word containing a pair of instructions has data type dtp-fix. The low 16 bits of the fixnum contain the low 16 bits of the first instruction, and the high 16 bits of the fixnum contain the low 16 bits of the second instruction. The high-order bit of each instruction is stashed in the cdr code.

A CCA (compiled-code-accessor) is an internal data structure used by various functions for manipulating compiled code. On the 3600 a CCA is a locative pointer to the header of compiled function. In the cross-compiler and cross-debugger, the same routines accept a special defstruct as their CCA argument.

The header is stored in the first memory word, not in the first instruction word. The following field-access functions expect the address of the header as their argument. A CCA may be used, but a dtp-compiled-function object reference may not be used; the names of the fields are misleading.

The header word has data type DTP-HEADER-I and header type %HEADER-TYPE-COMPILED-FUNCTION.
The 28-bit immediate field of the header word contains:

COMPILED-FUNCTION-TABLE-SIZE	the number of words in the external reference table
COMPILED-FUNCTION-TOTAL-SIZE	total number of words in the structure
COMPILED-FUNCTION-SPARE-BITS-1	four bits that are not currently used

The fields in the remaining three words of the prefix are:

COMPILED-FUNCTION-ARGS-INFO	value to be returned by %ARGS-INFO
COMPILED-FUNCTION-SPARE-BITS-2	fourteen bits that are not currently used
COMPILED-FUNCTION-EXTRA-INFO	an object reference to a cons described below
COMPILED-FUNCTION-FUNCTION-CELL	a function cell described below

The extra-info is a cons whose car is the function name and whose cdr is the debug-info alist. All the data not needed during normal execution are in here, along with information needed by the compiler for substs and information needed by the interpreter for special forms with a more complicated argument pattern than just a quoted &rest arg.

[Describe debug-info better here? See the manual, which is pretty far out of date.]

The value returned by %ARGS-INFO is described in the manual and in 3600/LM-2 compatibility notes [whatever its proper title is, I don't have a copy.]

The function cell is used to improve virtual-memory locality. When the function definition of a symbol is a compiled function, the symbol's function cell is forwarded (with dtp-one-q-forward) to the function cell in the prefix of the compiled function, and all references to the symbol's function cell that can be found are changed to point directly to the compiled function's function cell. This function cell normally contains an object reference to the compiled function that contains it, but if the symbol is later redefined, the function cell will contain the new definition. It is also possible for the function cell to contain an unbound marker instead of an object reference. This function cell scheme avoids the need to keep a symbol paged into main memory when the symbol is only being used as the name of a function and the function is only being called from compiled code.

The first instruction in a compiled function is special; it is not executed the same way as all other instructions and does not contain a normal opcode. Instead the entry instruction contains the following fields. Since only 16 bits of the entry instruction are used, and they are in the low 16 bits of the first instruction word, these fields can also be regarded as fields of the first instruction word, the word pointed to by the address in an object reference to a compiled function.

%%ENTRY-INSTRUCTION-TABLE-SIZE	Number of words in the external reference table
%%ENTRY-INSTRUCTION-ARGS-DISPATCH	Dispatch code based on the number of arguments

%%ENTRY-INSTRUCTION-TABLE-SIZE is a copy of COMPILED-FUNCTION-TABLE-SIZE. Storing this information twice allows either reference point in a compiled function to be found quickly given the other reference point.

%%ENTRY-INSTRUCTION-ARGS-DISPATCH is a 4-bit code looked up in a microcode table along with the number of arguments supplied to determine how to enter the function. A value of zero here means the "slow" case is to be used; the function contains instructions from the TAKE-ARG family that will check the number of arguments and receive the arguments. Non-zero values encode all cases involving zero to four arguments, any number of which may be optional, and no rest or keyword arguments. The number of arguments is checked, the arguments that were supplied are copied into the function's frame, and the instructions initialize optional arguments that are not missing are skipped, all under the control of this dispatch. In a method, the two extra arguments (self and self-mapping-table) are included in the arguments described by this dispatch.

>> Closures dtp-closure

A closure is represented with list representation, for no particularly good reason. This does allow closures to be stored in the stack (a la WITH-STACK-LIST);

certain special forms such as ERROR-RESTART exploit this.

The list is always cdr-coded, but nothing actually depends on this. The first element of the list is the function. Succeeding elements are taken in pairs. The first element of each pair is a locative pointer to the value cell to be bound when the closure is called. The second element of each pair is a locative pointer to the closure value cell to which that call is to be linked.

> Non-object-reference data types

These data types identify special memory words. They may not appear in object references. Special memory words cannot be the values of variables, cannot be arguments to functions, and cannot be returned by functions. Special memory words may be divided into three categories: unbound markers, headers, and invisible pointers.

DTP-NULL labels an unbound marker. These are used in value cells and function cells of symbols, in instance-variable value cells in instances, in the function cell in a compiled function, and in closure value cells. The address field contains the address of the symbol that names the cell that is unbound.

DTP-NULL is also used to initialize freshly-created virtual memory in case it is accidentally accessed before an object is created in it. The address field contains the word's own address.

The value of DTP-NULL is zero; thus memory that is initialized to all bits zero (the FEP command Clear Machine does this) contains DTP-NULL words, which will cause a trap if referenced.

DTP-HEADER-P and DTP-HEADER-I label headers. These were described above.

There are several types of invisible pointer. These are also known as forwarding pointers because they "forward" a memory reference from one memory location to another, by analogy with postal forwarding. Most of these types are adequately described in section 14.2 of the manual.

DTP-EXTERNAL-VALUE-CELL-POINTER is not invisible to binding and unbinding. It is used to link a symbol's value cell to a closure or instance value cell.

DTP-ONE-Q-FORWARD forwards only the cell that contains it. It is used to link a symbol value or function cell to a wired cell, an A-memory cell, or a compiled-function's function cell, as well as many other applications. [ugh]

DTP-HEADER-FORWARD is used when a whole structure is forwarded. This word marks where the header used to be, and contains the address of where the header is now. The other words of the structure are forwarded with DTP-ELEMENT-FORWARD. DTP-HEADER-FORWARD is also used in connection with list representation. When a 1-word cons must be expanded to a 2-word cons by RPLACD, a new 2-word cons is allocated and the old 1-word cons is replaced by a DTP-HEADER-FORWARD containing the address of the new cons. The cdr code in the location containing the forwarding point is ignored.

DTP-ELEMENT-FORWARD is used when a whole structure is forwarded. The address field contains the address of where the word that used to be here was moved to. Every word except the header contains a DTP-ELEMENT-FORWARD.

DTP-BODY-FORWARD is an obsolete left-over from the LM-2, no longer used. See below.

DTP-GC-FORWARD is used by the garbage collector and may only appear in oldspace. When an object is evacuated from oldspace, each word of the object's former representation contains a DTP-GC-FORWARD that points to the new location of that word.

DTP-MONITOR-FORWARD is not yet implemented. It is intended to be used in the future for a debugging feature that allows modifications to a particular storage location to be intercepted. cf. the LM-2 MONITOR-VARIABLE semifeature.

>> The difference between DTP-BODY-FORWARD and DTP-ELEMENT-FORWARD:

When a structure is copied, e.g. by ADJUST-ARRAY-SIZE, and the old copy is forwarded to the new copy, the header word of the old structure is clobbered with a DTP-HEADER-FORWARD that points to the header word of the new structure. Each other word of the old structure is clobbered with a DTP-ELEMENT-FORWARD that points to the corresponding word of the new structure. The old→new mapping may not be straightforward if the structure has been rearranged during the copying process.

To find the size of the old version of a structure that has been copied, count the number of DTP-ELEMENT-FORWARDs that follow the

DTP-HEADER-FORWARD. The search will be terminated by the header (forwarded or not) of the next structure or by the end of the allocated part of the region.

DTP-BODY-FORWARD is a holdover from the LM-2. It can be removed eventually (in Release 6 perhaps). When a structure is copied, each word of the old structure other than the header is clobbered with a DTP-BODY-FORWARD that points to the old header word, which in turn points to the new header word. The corresponding word of the new structure is at the same offset from the new header word as the word of the old structure was from the old header.
DTP-BODY-FORWARD is needed on the LM-2 because the header of an array is not necessarily in its first storage location, which means that finding the size of the old version of a copied structure has to be able to distinguish body-forwards belonging to the old structure from body-forwards belonging to the next-higher structure in memory, if it, too, is forwarded.

> Address-space organization

Address space is divided into four major parts. In order of increasing addresses, they are:

permanently-wired space

normal virtual space (portions of which can be wired)

physical address space

A-memory address space

Permanently-wired space contains two communication areas, the Lisp macrocode that handles the disk, the Lisp macrocode for paging, and the various tables needed by that macrocode. Permanently-wired space is initialized by the FEP Load World command from an image in the world load file, and contains everything needed to page in the rest of the world load. Permanently-wired space starts at virtual address 0 and ends at %WIRED-VIRTUAL-ADDRESS-HIGH. It occupies a contiguous block of physical address space starting at %WIRED-PHYSICAL-ADDRESS-LOW. Variable-sized tables needed by the paging routines are allocated from physical space.

Normal virtual space occupies most of the address space. Its organization is described below.

Physical address space occupies the uppermost 1/16 of the address space, and allows every location on the Lbus to be addressed. This is useful for accessing memory-mapped I/O devices on the Lbus. It can also be used to access main memory directly by physical address; this is used for certain wired tables. The paging and disk code does not reside in physical space, because the physical addresses of usable memory are not fixed; the system must be able to boot on a machine with no memory board in Lbus slot 0. This is why permanently-wired space exists. See the byte field %%VMA-EQUALS-PMA described below.

The top 4096 words of address space are the A memory in the processor data path. These locations contain several communication areas, and also contain the auxiliary stack buffer. The main stack buffer physically exists in this space, but is normally addressed by virtual addresses. When a page of a stack is moved from main memory into the stack buffer, the mapping of its virtual address is changed so that it maps to the physical address of the appropriate page in A memory rather than the physical address of a page in main memory.

>> Communication areas

There are several tables at fixed addresses that are used for communications between independently-maintained parts of the system. The communication entities are the Lisp system, the microcode, the FEP, and the remote debugger (which operates through the FEP). The microcode, the FEP, and the remote debugger need to be able to find objects in the Lisp world; the addresses of these objects vary from system to system. The usual way that communication areas are accessed from Lisp is that the value cell of each symbol that names a communication location is forwarded (with DTP-ONE-Q-FORWARD) to that location. Thus the communication area locations are simply accessed as special variables. The addresses of communication-area locations are known by the microcode, the Lisp world, the FEP, and the remote debugger, so they cannot be moved without a coordinated simultaneous installation of new versions of each of these pieces of software. Effectively this means that they can never be moved.

The details of each communication area are given in the file SYS: L-SYS; SYSDF1 LISP. They will not be repeated here, except for the highlights.

The FEP-COMMUNICATION-AREA is located at virtual address 0 and at physical address %WIRED-PHYSICAL-ADDRESS-LOW. It is one page long. This block of memory is initialized to a "null" state, mostly unbound markers, by the world load file and is then filled in by the FEP. Everything that is allocated in physical memory by the FEP has its address stored here, so that Lisp can find it. In addition, the main locations through which Lisp and the FEP communicate while the system is running are here.

The SYSTEM-COMMUNICATION-AREA immediately follows the FEP-COMMUNICATION-AREA in virtual and physical addresses, and is also one page long. This block is initialized from the world load file. It contains the addresses of all the objects and tables that are needed by the FEP or by the remote debugger to find their way around in Lisp virtual memory. When the remote debugger is examining a world load file it uses the image of the SYSTEM-COMMUNICATION-AREA in that file to find its way around. In general, this communication area contains the version number of the system; addresses of the paging system's tables; addresses of the area, region, and quantum tables; the address of the current package (so that the remote debugger can look up Lisp symbols); locations for keyboard and mouse input from the FEP; and the useful symbols NIL, T, and IGNORE.

The A-MEMORY-VARIABLES are a communication area used to communicate between the microcode and Lisp. The FEP and the remote debugger also look at a few of these locations, but only when the machine is stopped (the FEP cannot access A-memory while the machine is running). This block is initialized by the microcode load file. The A-memory variables include a set of variables for each module of the system that happens to be split between microcode and macrocode. To summarize, variables exist for the stack buffer, the virtual memory, the disk, the network, the audio output, the 1/2" tape drive, sequence breaks, character comparison case-dependence, arithmetic, the garbage collector, object allocation, and metering features.

Other communication areas in A-memory include MICROCODE-CONSTANTS, which supplies the microcode with the addresses of various Lisp objects and Lisp functions; MICROCODE-ESCAPE-ROUTINES, which supplies the microcode with the addresses of hand-coded macrocode sequences that it sometimes requires [this should be explained in detail somewhere]; CURRENT-STACK-GROUP-DATA, a copy of various information about the currently-executing stack group, encached in A-memory so it can be accessed more quickly than if it were in main memory; OTHER-STACK-GROUP-DATA, the same information for the stack group executing in the main stack buffer, used when control is in the auxiliary stack buffer; MICROCODE-ESCAPE-CONSTANTS, constants used by the MICROCODE-ESCAPE-ROUTINES.

>> Organization of Virtual Memory

The normal virtual memory, along with the portion of the permanently-wired space after the two communication areas at the front, is used to store Lisp objects. Therefore this portion of the machine's address space must be organized for dynamic allocation of storage as new objects are created, and the parts that are allocated to objects must be subject to the conventions described above (subsection Storage Conventions).

Virtual memory is divided up in five different ways. The units of division are called pages, areas, spaces, regions, and quanta.

A page is 256 words long. Pages are the units of transfer between the disk and main memory. The demand-paged virtual memory deals in pages. Higher levels of the software usually do not deal specifically with pages, except when they have to wire down an object (usually in order to do I/O) so that it stays in main memory and does not get paged out. Wiring down is done in units of pages.

An area is a user-defined portion of virtual memory that contains objects used for a specific purpose. When new objects are created, the area in which they are to be created must be specified, or defaulted. An area does not have a fixed size and need not be a single contiguous range of addresses. An area is not represented by a Lisp object, but instead by a name (a symbol) and a number (a small integer starting from zero). The reasons for this are lost in antiquity. See the Lisp Machine Manual for more about areas; they are a language feature rather than strictly architecture.

A space is a portion of virtual memory that is treated in a particular way by the storage allocation system, including the garbage collector. An area may contain storage that is part of more than one space, and nearly all spaces exist in more than one area. The following space types exist:

FREE	not being used for anything. Other spaces can expand by gobbling up part of FREE space.
OLD	contains condemned objects that will go away if they are not still referenced. Any condemned object that can still be reached is moved into COPY space, and then OLD space is reclaimed, i.e. becomes FREE.

NEW	newly-created objects are allocated here.
COPY	condemned objects evacuated from oldspace are allocated here by the transporter.
STATIC	newly-created objects that are very unlikely to become garbage are allocated here. It is an attribute of an area that controls whether new objects allocated in that area are placed in NEW space or STATIC space.
STACK	control stacks of stack groups are allocated here. Stacks require special treatment because the portion of a stack that contains references to objects that the stack really cares about varies as the stack pointer moves up and down.

A region is a contiguous range of addresses that lies entirely in one space and entirely in one area and contains objects of one representation type (list or structure). Regions are the building-blocks out of which areas are constructed. Since a region is the largest range of addresses that is to be treated in a uniform way by the storage allocation system, it is convenient to organize many of that system's tables around regions. A region is identified by a number, which is an index into the tables of information about regions.

A quantum is 16384 words long. The length of a region is always an integral multiple of a quantum. Thus quanta are the units of allocation of storage to regions. The number of quanta in the entire address space is of manageable size (also 16384). This permits a couple of important tables whose lookup must be extremely fast to be indexed by quantum (see below).

Words are the unit of allocation of storage to objects.

The area WIRED-CONTROL-TABLES, which is always area number zero, has one region, always region number zero. It is in the STATIC space and contains objects in structure representation. This area consists of the permanently-wired space containing the paging and disk software and the FEP and SYSTEM communication areas. WIRED-CONTROL-TABLES contains some extra space and new objects can be allocated in it. This provides a useful way to patch the permanently-wired kernel of the system. Once allocation in WIRED-CONTROL-TABLES crosses a page boundary, however, special measures must be taken to make permanently-wired space larger. Currently this requires patching the world load file to declare a larger permanently-wired space and then cold booting.

>> Area and Region Tables

The tables described in this section are all accessible through the system communication area, as well as by Lisp variables or functions as described. Some of these tables are stored in permanently-wired space and accessed by the paging system.

The value of ADDRESS-SPACE-MAP is an array, indexed by quantum number, whose value is the region number of the region occupying that quantum, or zero if the quantum is free. Zero can also mean that the quantum is occupied by region zero, the WIRED-CONTROL-TABLES.

The GC-map is an array, indexed by quantum number, whose value is a 3-bit code describing the space occupying that quantum. This array resides in hardware on the datapath board and controls the hardware that assists the garbage collector.

The following area tables exist. Each of these is an array that resides in the function definition cell of the specified symbol, indexed by area number. Thus these tables can be regarded as functions from area numbers to information about the area. SETF is used to store into these tables.

AREA-NAME	The symbol that names the area
AREA-MAXIMUM-SIZE	Maximum number of words to allocate to area
AREA-REGION-SIZE	Number of words desired per region
AREA-REGION-BITS	Initial value for REGION-BITS of regions in this area
AREA-REGION-LIST	Number of first region (not a Lisp list!)

The following region tables exist. Each of these is an array that resides in the function definition cell of the specified symbol, indexed by region number. Thus these tables can be regarded as functions from region numbers to information about the region. SETF is used to store into these tables.

REGION-ORIGIN	Address of start of region (a fixnum)

REGION-LENGTH	Number of words allocated to the region
REGION-FREE-POINTER	Number of words actually in use by objects
REGION-GC-POINTER	Number of words scanned by (long-term) gc
REGION-BITS	Fixnum of random fields (see below)
REGION-LIST-THREAD	Number of next region in this area

The origin and length of a region are always multiples of the quantum size.

The free pointer is the address, relative to the origin of the region, at which the next object can be allocated. The length minus the free pointer is the number of words remaining for allocation of objects.

All of the regions in an area are threaded together through REGION-LIST-THREAD. The head of the list is in AREA-REGION-LIST. The last element of the list contains a fixnum which is the area number with the sign bit forced on; thus the list is circular. The %AREA-NUMBER subprimitive uses ADDRESS-SPACE-MAP to find the region number and then follows the region list thread until it gets back to the area. Note that the list is terminated by (LOGIOR AREA (ROT 1 -1)), not by (MINUS AREA).

The fields in REGION-BITS are as follows. Each field is given as the symbolic name for its byte specifier, followed by the symbolic names (if any) for the values it may take on.

```
    %%REGION-REPRESENTATION-TYPE    A code for the object representation type
            %REGION-REPRESENTATION-TYPE-LIST
            %REGION-REPRESENTATION-TYPE-STRUCTURE

    %%REGION-SPACE-TYPE             A code for the space
            %REGION-SPACE-FREE
            %REGION-SPACE-OLD
            %REGION-SPACE-NEW
            %REGION-SPACE-COPY
            %REGION-SPACE-STATIC
            %REGION-SPACE-STACK

    %%REGION-SCAVENGE-ENABLE        If 1, scavenger scans object references from the
                                    allocated part of this region, since they could
                                    be to objects in old space.

    %%REGION-READ-ONLY              If 1, write-protect pages in this region

    %%REGION-SWAPIN-QUANTUM         Number of pages at a time to read from disk

    %%REGION-TEMPORARY              OK to reset this region with reset-temporary-area

    %%REGION-PAGING-TYPE            Different paging styles
            %PAGING-TYPE-NORMAL
            %PAGING-TYPE-SEQUENTIAL

    %%REGION-SAFEGUARD              1 if region may not be condemned because it
                                    contains part of the garbage collector or
                                    part of the storage system, i.e. objects that
                                    the transporter depends on in order to run.

    %%REGION-EPHEMERAL              1 if region contains ephemeral data
                                    and all pointers into it are page-tagged (NYI)

    %%REGION-LEVEL                  Level of ephemeralness (counts down to zero)
                                    In oldspace this is what copyspace to use

    %%REGION-NO-CONS                1 if region not available for allocation
                                    of new objects
```

>> Byte fields and constants relating to addresses

PAGE-SIZE	256.	The number of words in a page
%%VMA-PAGE-NUM	20. 8	The virtual page number field of an address
%%VMA-WORD-OFFSET	8 0	The word number within a page
%%VMA-EQUALS-PMA	4 24.	If these bits are 1's, the address is physical
%%VMA-EQUALS-AMEM	16. 12.	If these bits are 1's, the address is in A-memory
%%VMA-QUANTUM-NUM	14. 14.	The quantum number field of an address
%ADDRESS-SPACE-QUANTUM-SIZE	16384.	The number of words in a quantum
A-MEMORY-VIRTUAL-ADDRESS	1777770000	The virtual address of the first location in A-memory
MAIN-STACK-BUFFER-ADDRESS		The virtual address of the first A-memory

	location used for the main stack buffer
AUXILIARY-STACK-BUFFER-ADDRESS	The virtual address of the first A-memory location used for the auxiliary stack buffer

> Stacks

The state of a computation is partially expressed in three stacks: the control stack, the binding stack, and the data stack. Memory locations and objects whose lifetime is known to end when a function returns or is thrown through are allocated in stacks, rather than in the main object storage system, to increase efficiency. The control stack contains function-nesting information, arguments, local variables, function return values, and miscellaneous control information. The binding stack records bindings of special variables and other memory locations. The data stack contains stack-allocated temporary objects (it is not yet implemented).

The primary design criterion for the stacks was to make function calling as fast as possible, even at the expense of making stacks bigger than the minimum possible size. The stack buffer will largely eliminate any performance deficit from large frame headers in the control stack, thus packing of information into bytes (as in the CADR) does not make sense by and large. Function calling is probably the one thing in the system most critical to performance of cpu-bound (non-paging) programs. It is important to make the overhead as small as possible without compromising features. Other design criteria are to provide for lexical scoping and nested functions (not yet implemented), to provide the ability for the debugger to obtain complete information about the state of the program, even when running compiled functions, and to implement the &optional and &rest argument features, the catch/throw/unwind-protect feature, and the multiple-value return feature efficiently.

>> Stack groups

A stack group is a named structure (a type of array) that contains information about the state of a computation, including its three stacks. See the Lisp Machine Manual for the high-level conceptual description and the function that create and manipulate stack groups.

The fields in a stack group are:

Locative pointers to various locations in the stacks:

SG-FRAME-POINTER	Current (i.e. innermost) frame in control stack
SG-STACK-POINTER	Current top of control stack
SG-CONTROL-STACK-LOW	First word in control stack
SG-CONTROL-STACK-LIMIT	Highest address at which a frame may be created
SG-BINDING-STACK-LOW	First word in binding stack
SG-BINDING-STACK-LIMIT	Highest legal value of SG-BINDING-STACK-POINTER
SG-BINDING-STACK-POINTER	Current top of binding stack
SG-CATCH-BLOCK-LIST	Innermost catch block, or NIL if there is none

Fixnums containing miscellaenous fields and bits:

SG-FLOAT-OPERATING-MODE	Value of the variable FLOAT-OPERATING-MODE; this enables various features of IEEE floating point
SG-FLOAT-OPERATION-STATUS	Value of the variable FLOAT-OPERATION-STATUS; this records various IEEE floating point exceptions and conditions that have occurred
SG-STATUS-BITS	A word containing the rest of the fields:
SG-ARG-STATUS	Call/return/argument status code. Symbolic names for possible values of this field are:
%SG-ARG-NONE	No argument desired (e.g. has been preset)
%SG-ARG-BREAK	Interrupted or trapped, no argument desired
%SG-ARG-RESUME	Exited via STACK-GROUP-RESUME, argument wanted
%SG-ARG-CALL	Exited by calling another stack group, arg wanted
%SG-ARG-RETURN	Exited via STACK-GROUP-RETURN, arg wanted
SG-NONRESUMABILITY	Bits that prevent the stack group from being resumed:
SG-ACTIVE-BIT	1 if it is currently executing on the machine
SG-EXHAUSTED-BIT	1 if the initial function returned
SG-PROCESSING-ERROR-FLAG	1 if in initial error processing
SG-UNINITIALIZED-BIT	1 if no computation has been preset into it yet
SG-SAFE	1 if "safe" call/return mode was specified
SG-NONTRAPPABILITY	Bits that cause the microcode to halt if an error trap from microcode to macrocode occurs:
SG-STACK-LOAD-STARTED	Transfer between stack buffer and memory in progress
SG-HALT-ON-ERROR	Set when trap starts, cleared when it completes and condition signalling begins. This detects recursive

	errors that would otherwise loop infinitely.
SG-METER-ENABLE	Set to enable function entry/exit metering

Other fields used only by Lisp code, not by the microcode:

SG-NAME	A string
SG-FOOTHOLD-DATA	For use by error handler and debugger (not used?)
SG-PREVIOUS-STACK-GROUP	Resumer (when called as a function)
SG-ARGLIST	List of arguments (when called as a function)
SG-DATA-STACK-LOW	NIL if no data stack, else locative to first word
SG-DATA-STACK-LIMIT	Highest address at which a frame may be created
SG-DATA-STACK-POINTER	Locative to highest valid word
SG-DATA-STACK-FRAME-POINTER	Locative to highest frame

Note that the references to call/return status above refer to calling a stack group as a function and returning with STACK-GROUP-RETURN; they do not refer to ordinary function calling within the stack group.

>> Control Stack

--- here begins stuff not done yet

stack frames
the stack buffer
maximum frame size
method calling
lexpr calling

an old paragraph about argument copying:

Unlike the previous version of this L-machine architecture, this version abandons the CADR's inverted calling sequence. This is done in order to improve the speed of function calling, and to simplify the compiler. The reason for the inverted calling sequence in the CADR architecture was to avoid the overhead of copying the arguments; it arranges that when the caller pushes the arguments on the stack they are in exactly the place where the callee wants them. However, it turns out that for functions with less than about 7 arguments, the overhead for copying the arguments is less than the other overheads associated with the inverted calling sequence. Also lexical scoping is simpler with the non-inverted calling sequence. Hence this architecture proposal.

```
;;; Catch blocks

;Each stack-group has a threaded list of catch blocks in its control stack.
;These blocks are used for unwind-protect also.

(DEFSTORAGE (CATCH-BLOCK)
        (CATCH-BLOCK-TAG)               ;The tag being caught
                ;Temporarily we will use the symbol T to mean unwind-protect
        (CATCH-BLOCK-PC)                ;PC of end of catch body, or of cleanup handler
        (CATCH-BLOCK-BINDING-STACK-POINTER)     ;To unwind special-variable bindings
        ((CATCH-BLOCK-PREVIOUS)         ;NIL or locative to previous block in this sg
         (CATCH-BLOCK-VALUE-DISPOSITION 2 34.)))        ;Same as FRAME-VALUE-DISPOSITION
                ;The value disposition is used only when the catch is thrown to,
                ;not when the body is exited normally.  Thus the microcode doesn't
                ;process it, only the THROW function does. (The internal function
                ;that implements the THROW special form.)
```

>> Binding Stack

Any memory location can have its contents temporarily changed while control is inside of a particular construct. This is called binding the location. The temporarily changed value is visible from all functions called by the code in the body of the construct (binding has dynamic scope), but not by the execution of other stack groups.

The memory locations most commonly bound are the value cells of special variables. These can be bound by such constructs as the LET special form. However, any memory location can be bound using the BIND subprimitive, and it is not uncommon to see communication-area locations (especially in A-memory), function-definition cells of symbols, instance variables, or array elements being bound. Any memory location that can contain an object reference can be bound.

Binding is "shallow," i.e. it operates by actually replacing the contents of the location, remembering the old contents in an entry in the binding stack. When the scope of the binding is exited, the binding-stack entry is popped off and the old contents of the location are restored. When control switches from one stack group to another, bindings are swapped: To leave the context of one stack group the "outside" values remembered in the binding stack are restored and the "inside" values are saved in the binding stack. To enter the context of another stack group, the "inside" values are retrieved from the binding stack and the "outside" values are saved in the binding stack.

Binding never affects the cdr code of the memory location bound, only the contents (an object reference). The reads and writes associated with binding follow forwarding pointers, except for DTP-EXTERNAL-VALUE-CELL-POINTER pointers, which are treated as data. DTP-NULL pointers (unbound markers) are also treated as data and do not cause a trap.

The binding stack consists of a sequence of 2-word entries. Zero or more entries are associated with each frame in the control stack. The fields in an entry are as follows:

BINDING-STACK-CELL	A locative pointer to the memory cell that is bound
BINDING-STACK-CHAIN-BIT	1 if the previous entry is for the same frame
BINDING-STACK-CLOSURE-BIT	1 if this binding was created by a closure or instance
BINDING-STACK-CONTENTS	Saved contents of bound cell

BINDING-STACK-CONTENTS is the "outside" or "former" contents of the cell if the stack group that owns this binding stack is active. Otherwise it is the "inside" or "current" contents of the cell.

If BINDING-STACK-CLOSURE-BIT is set, the binding was created before the function was entered, and should not be undone if the function is retried (via the c-m-R command in the debugger).

Note that an entry is designated by a pointer to its last word, so that the binding-stack-pointer designates the innermost entry. BINDING-STACK-CLOSURE-BIT and BINDING-STACK-CHAIN-BIT are in the cdr-code field of one of the two words.

--- here begins stuff not done yet

>> Data Stack

not yet implemented

>> Lexical closures

not yet implemented

> Auxiliary Stack Buffer

--- here begins stuff not done yet

page faults
disk interrupts
sequence breaks

> Macroinstructions

--- here begins stuff not done yet

design principles
format
addressing modes

> Page Tags

Date: Wednesday, 22 February 1984, 21:08-EST
From: David A. Moon <Moon at SCRC-TENEX>
Subject: Page modified tags.
To: David C. Plummer <DCP at SCRC-TENEX>
Cc: I-Architecture at SCRC-TENEX, DanG at SCRC-TENEX
In-reply-to: The message of 22 Feb 84 01:37-EST from David C. Plummer <DCP at SCRC-TENEX>

> Date: Wednesday, 22 February 1984, 01:37-EST
> From: David C. Plummer <DCP at SCRC-TENEX>
>
> The 3600 has page referenced tags to assist the storage system in finding pages that haven't been touched and are therefore better candidates for others to simply discard.
>
> I don't know if the following is appropriate for the I-Machine, but here goes... What about page modified tags? Currently, lisp simulates them by setting newly fetched pages to take a page fault on write.

There is an important difference between page-modified tags and page-referenced tags. Every change of a page-modified tag from 0 to 1 implies an eventual disk operation to write out that modified page (except for the modified pages that are still in core at the end of the day when you halt the machine and go home). The cost of the "write-first" page fault that marks the page as modified is dwarfed by the cost of the disk operation.

Page-referenced tags, on the other hand, change from 0 to 1 much more frequently than the rate of page fetches, with the ratio being related to the size of the working set. (Lisp keeps clearing the page-referenced tags and then if they don't get set again after a while, the page is not being used and is a good candidate for replacement.)

Thus special hardware is much more effective for the page-referenced tags than for the page-modified tags. Here's a good joke for all you architects out there: The VAX has page-modified tags but not page-referenced tags.

Beginner Documentation:

System Overview

Front End Processor(FEP)

(1)The FEP loads microcode into the control memory (CMEM) on the Sequencer via the SPY bus from the hard disk.

(2)The FEP board is home for the Cartridge Tape Drive, the three serial ports and keyboard interface (which, incidentally, the mouse is mapped into).

(3)This board is home for the high speed buffer for the hard disk which has its SMD interface built into it.

(4)When the FEP first comes up it:
- (a)The 68000 reads in its EPROM firmware and
  - (1)initializes RAM, Serial I/O
  - (2)loads microcode into Sequencer

(5)There are three clocks on the FEP
- (a)The 16mhz oscillator which is clock for the 68000
- (b)The 4.92mhz oscillator which is the serial ports clock source for baud rates.
- (c)The 66.67mhz oscillator which is the source of main overall system timing.

(6)When the FEP comes up it initializes the GC(garbage collector) on the Data Path board and some stuff to do with A-MEM.

Input/Output Board(I/O)

(1)The I/O board is home to the following:
- (a)The console video
  - (1)Video Ram
  - (2)Vertical and horizontal sync
  - (3)ECL-->video data out
  - (4)80mhz oscillator for video
- (b)The Ethernet interface and its crc(error checking)
- (c)The Disk Status Register(ECC, shift registers out to L-bus)

Data Path Board(DP)

(1)The Data Path board contains the following elements
- (a)A single chip multiplier
- (b)Two RAM stacks, A-MEM and B-MEM which are the temporary variable storage areas equivalent to general purpose registers in more conventional systems. B-MEM is initialized once at system startup (locations 0->360 no R/W, 360->377 used as a stack.
- (c)The Frame Pointer which points into the stack at the microcode subroutine return address.
- (d)The Stack Pointer which points into the A-MEM stack.
- (e)The DP does alot of address calculation and address supplying for virtual memory paging of the disk.
- (f)The DP communicates to the Memory Controller via the MCA bus.
- (g)A-MEM has its own address adder calculation circuitry, A-MEM will often supply the starting address of macrocode.
- (h)Byte swapping is provided by gating data from the A or B stacks through the X and Y buses.

Memory Controller Board(MC)

(1)The Memory Controller board is where all accesses to memory are handled. It supports the Virtual Memory paradigm where the disk space is made to appear as an extension of the ram address space via physical->logical software configured mapping. In hardware this is accomplished through a hierarchical system of pointers and buffers.Other elements of this board are as follows:
- (a)The MC does the ECC (Error Correction Code)calculations on data fetched from memory.
- (b)The Instruction Prefetch (4 instructions) to I-Buffers lives here.
- (c)The macrocode program counter and the counters for the Virtual Memory Address(VMA) are resident here.
- (d)The IFU Dispatch (Instruction Fetch Unit) which determines the next Cmem (Control Memory) address of microcode to be executed is here.
- (e)The EMU-MD pair registers (74f373's) which support the main data transfer of most everything are located here. The acronym EMU-MD stands for Emulator-Memory Data.

```
        -*- Mode: Text -*-   Notes on updating this file:

This glossary file is being maintained at two main locations.  It is
AIWORD.RF[UP,DOC] at SAIL, and GLS;JARGON > at MIT.  If you make any
changes, be sure to FTP the new file to the other location.   (NOTE:
Use ASCII mode in FTP to  avoid screwing up the tilde char!)   It is
a good idea to merge changes  with the version on  the other machine
in case other people forgot to do the FTP.  Also, please let us know
(see list of names in later paragraph) about your changes so that we
can double-check them.

Try to conform to the format already being used--70 character lines,
3-character indentations, pronunciations in parentheses, etymologies
in brackets, single-space after def'n numbers and word classes, etc.

Stick to the standard ASCII character set.

If you'd rather not mung the file yourself, send your definitions to
DON @ SAIL, GLS @ MIT-AI, and/or MRC @ SAIL.

The last edit (of this line, anyway) was by Phil Agre, 6/26/82.

======================================================================

        Compiled by Guy L. Steele Jr., Raphael Finkel, Donald
        Woods, and Mark Crispin, with assistance from the MIT
        and Stanford AI communities and Worcester Polytechnic
        Institute.  Some contributions were submitted via the
        ARPAnet from miscellaneous sites.

Verb doubling: a standard construction is to double a verb and use it
   as a comment on what the implied subject does.  Often used to
   terminate a conversation.  Typical examples involve WIN, LOSE,
   HACK, FLAME, BARF, CHOMP:
        "The disk heads just crashed."  "Lose, lose."
        "Mostly he just talked about his --- crock.  Flame, flame."
        "Boy, what a bagbiter!  Chomp, chomp!"

Soundalike slang: similar to Cockney rhyming slang.  Often made up on
   the spur of the moment.  Standard examples:
        Boston Globe => Boston Glob
        Herald American => Horrid (Harried) American
        New York Times => New York Slime
        historical reasons => hysterical raisins
        government property - do not duplicate (seen on keys)
                => government duplicity - do not propagate
   Often the substitution will be made in such a way as to slip in
   a standard jargon word:
        Dr. Dobb's Journal => Dr. Frob's Journal
        creeping featurism => feeping creaturism
        Margaret Jacks Hall => Marginal Hacks Hall

The -P convention: turning a word into a question by appending the
   syllable "P"; from the LISP convention of appending the letter "P"
   to denote a predicate (a Boolean-values function).  The question
   should expect a yes/no answer, though it needn't.  (See T and NIL.)
     At dinnertime: "Foodp?"  "Yeah, I'm pretty hungry." or "T!"
     "State-of-the-world-P?"  (Straight) "I'm about to go home."
                              (Humorous) "Yes, the world has a state."
   [One of the best of these is a Gosperism (i.e., due to Bill
   Gosper).  When we were at a Chinese restaurant, he wanted to know
   whether someone would like to share with him a two-person-sized
   bowl of soup.  His inquiry was: "Split-p soup?" --GLS]

Peculiar nouns: MIT AI hackers love to take various words and add the
   wrong endings to them to make nouns and verbs, often by extending a
   standard rule to nonuniform cases.  Examples:
                porous => porosity
                generous => generosity
        Ergo:   mysterious => mysteriosity
                ferrous => ferocity
   Other examples:  winnitude, disgustitude, hackification.

Spoken inarticulations:  Words such as "mumble", "sigh", and "groan"
   are spoken in places where their referent might more naturally be
   used.  It has been suggested that this usage derives from the
```

impossibility of representing such noises in a com link. Another expression sometimes heard is "complain!"

@BEGIN (primarily CMU) with @END, used humorously in writing to indicate a context or to remark on the surrounded text. From the SCRIBE command of the same name. For example:

```
@Begin(Flame)
Predicate logic is the only good programming language.
Anyone who would use anything else is an idiot. Also,
computers should be tredecimal instead of binary.
@End(Flame)
```

ANGLE BRACKETS (primarily MIT) n. Either of the characters "<" and ">". See BROKET.

AOS (aus (East coast) ay-ahs (West coast)) [based on a PDP-10 increment instruction] v. To increase the amount of something. "Aos the campfire." Usage: considered silly. See SOS.

ARG n. Abbreviation for "argument" (to a function), used so often as to have become a new word.

AUTOMAGICALLY adv. Automatically, but in a way which, for some reason (typically because it is too complicated, or too ugly, or perhaps even too trivial), I don't feel like explaining to you. See MAGIC. Example: Some programs which produce XGP output files spool them automagically.

BAGBITER 1. n. Equipment or program that fails, usually intermittently. 2. BAGBITING: adj. Failing hardware or software. "This bagbiting system won't let me get out of spacewar." Usage: verges on obscenity. Grammatically separable; one may speak of "biting the bag". Synonyms: LOSER, LOSING, CRETINOUS, BLETCHEROUS, BARFUCIOUS, CHOMPER, CHOMPING.

BANG n. Common alternate name for EXCL (q.v.), especially at CMU. See SHRIEK.

BAR 1. The second metasyntactic variable, after FOO. "Suppose we have two functions FOO and BAR. FOO calls BAR..." 2. Often appended to FOO to produce FOOBAR.

BARF [from the "layman" slang, meaning "vomit"] 1. interj. Term of disgust. See BLETCH. 2. v. Choke, as on input. May mean to give an error message. "The function '=' compares two fixnums or two flonums, and barfs on anything else." 3. BARFULOUS, BARFUCIOUS: adj. Said of something which would make anyone barf, if only for aesthetic reasons.

BELLS AND WHISTLES n. Unnecessary but useful (or amusing) features of a program. "Now that we've got the basic program working, let's go back and add some bells and whistles." Nobody seems to know what distinguishes a bell from a whistle.

BIGNUMS [from Macsyma] n. 1. In backgammon, large numbers on the dice. 2. Multiple-precision (sometimes infinitely extendable) integers and, through analogy, any very large numbers. 3. EL CAMINO BIGNUM: El Camino Real, a street through the San Francisco peninsula that originally extended (and still appears in places) all the way to Mexico City. It was termed "El Camino Double Precision" when someone noted it was a very long street, and then "El Camino Bignum" when it was pointed out that it was hundreds of miles long.

BIN [short for BINARY; used as a second file name on ITS] 1. n. BINARY. 2. BIN FILE: A file containing the BIN for a program. Usage: used at MIT, which runs on ITS. The equivalent term at Stanford is DMP (pronounced "dump") FILE. Other names used include SAV ("save") FILE (DEC and Tenex), SHR ("share") and LOW FILES (DEC), and EXE ("ex'ee") FILE (DEC and Twenex). Also in this category are the input files to the various flavors of linking loaders (LOADER, LINK-10, STINK), called REL FILES.

BINARY n. The object code for a program.

BIT n. The unit of information. "Bits" is often used simply to mean information, as in "Give me bits about DPL replicators".

BITBLT (bit'blit) 1. v. To perform a complex operation on a large block of bits, usually involving the bits being displayed on a bitmapped raster screen. See BLT. 2. n. The operation itself.

BLETCH [from German "brechen", to vomit (?)] 1. interj. Term of disgust. 2. BLETCHEROUS: adj. Disgusting in design or function. "This keyboard is bletcherous!" Usage: slightly comic.

BLT (blit, very rarely belt) [based on the PDP-10 block transfer instruction; confusing to users of the PDP-11] 1. v. To transfer a large contiguous package of information from one place to another. 2. THE BIG BLT: n. Shuffling operation on the PDP-10 under some operating systems that consumes a significant amount of computer time. 3. (usually pronounced B-L-T) n. Sandwich containing bacon, lettuce, and tomato.

BOGOSITY n. The degree to which something is BOGUS (q.v.). At CMU, bogosity is measured with a bogometer; typical use: in a seminar, when a speaker says something bogus, a listener might raise his hand and say, "My bogometer just triggered." The agreed-upon unit of bogosity is the microLenat (uL).

BOGUS (WPI, Yale, Stanford) adj. 1. Non-functional. "Your patches are bogus." 2. Useless. "OPCON is a bogus program." 3. False. "Your arguments are bogus." 4. Incorrect. "That algorithm is bogus." 5. Silly. "Stop writing those bogus sagas." (This word seems to have some, but not all, of the connotations of RANDOM.) [Etymological note from Lehman/Reid at CMU: "Bogus" was originally used (in this sense) at Princeton, in the late 60's. It was used not particularly in the CS department, but all over campus. It came to Yale, where one of us (Lehman) was an undergraduate, and (we assume) elsewhere through the efforts of Princeton alumni who brought the word with them from their alma mater. In the Yale case, the alumnus is Michael Shamos, who was a graduate student at Yale and is now a faculty member here. A glossary of bogus words was compiled at Yale when the word was first popularized (e.g., autobogophobia: the fear of becoming bogotified).]

BOUNCE (Stanford) v. To play volleyball. "Bounce, bounce! Stop wasting time on the computer and get out to the court!"

BRAIN-DAMAGED [generalization of "Honeywell Brain Damage" (HBD), a theoretical disease invented to explain certain utter cretinisms in Multics] adj. Obviously wrong; cretinous; demented. There is an implication that the person responsible must have suffered brain damage, because he should have known better. Calling something brain-damaged is really bad; it also implies it is unusable.

BREAK v. 1. To cause to be broken (in any sense). "Your latest patch to the system broke the TELNET server." 2. (of a program) To stop temporarily, so that it may be examined for debugging purposes. The place where it stops is a BREAKPOINT.

BROKEN adj. 1. Not working properly (of programs). 2. Behaving strangely; especially (of people), exhibiting extreme depression.

BROKET [by analogy with "bracket": a "broken bracket"] (primarily Stanford) n. Either of the characters "<" and ">". (At MIT, and apparently in The Real World (q.v.) as well, these are usually called ANGLE BRACKETS.)

BUCKY BITS (primarily Stanford) n. The bits produced by the CTRL and META shift keys on a Stanford (or Knight) keyboard. Rumor has it that the idea for extra bits for characters came from Niklaus Wirth, and that his nickname was 'Bucky'.
DOUBLE BUCKY: adj. Using both the CTRL and META keys. "The command to burn all LEDs is double bucky F."

BUG [from telephone terminology, "bugs in a telephone cable", blamed for noisy lines; however, Jean Sammet has repeatedly been heard to claim that the use of the term in CS comes from a story concerning actual bugs found wedged in an early malfunctioning computer] n. An unwanted and unintended property of a program. (People can have bugs too (even winners) as in "PHW is a super winner, but he has some bugs.") See FEATURE.

BUM 1. v. To make highly efficient, either in time or space, often at the expense of clarity. "I managed to bum three more instructions." 2. n. A small change to an algorithm to make it more efficient.

BUZZ v. To run in a very tight loop, perhaps without guarantee of getting out.

CANONICAL adj. The usual or standard state or manner of something. A true story: One Bob Sjoberg, new at the MIT AI Lab, expressed some annoyance at the use of jargon. Over his loud objections, we made a point of using jargon as much as possible in his presence, and eventually it began to sink in. Finally, in one conversation, he used the word "canonical" in jargon-like fashion without thinking.
Steele: "Aha! We've finally got you talking jargon too!"
Stallman: "What did he say?"
Steele: "He just used 'canonical' in the canonical way."

CATATONIA (kat-uh-toe'nee-uh) n. A condition of suspended animation in which the system is in a wedged (CATATONIC) state.

CDR (ku'der) [from LISP] v. With "down", to trace down a list of elements. "Shall we cdr down the agenda?" Usage: silly.

CHINE NUAL n. The Lisp Machine Manual, so called because the title is wrapped around the cover so only those letters show.

CHOMP v. To lose; to chew on something of which more was bitten off than one can. Probably related to gnashing of teeth. See BAGBITER. A hand gesture commonly accompanies this, consisting of the four fingers held together as if in a mitten or hand puppet, and the fingers and thumb open and close rapidly to illustrate a biting action. The gesture alone means CHOMP CHOMP (see Verb Doubling).

CLOSE n. Abbreviation for "close (or right) parenthesis", used when necessary to eliminate oral ambiguity. See OPEN.

COKEBOTTLE n. Any very unusual character. MIT people complain about the "control-meta-cokebottle" commands at SAIL, and SAIL people complain about the "altmode-altmode-cokebottle" commands at MIT.

COM MODE (variant: COMM MODE) [from the ITS feature for linking two or more terminals together so that text typed on any is echoed on all, providing a means of conversation among hackers] n. The state a terminal is in when linked to another in this way. Com mode has a special set of jargon words, used to save typing, which are not used orally:

BCNU	Be seeing you.
BTW	By the way...
BYE?	Are you ready to unlink? (This is the standard way to end a com mode conversation; the other person types BYE to confirm, or else continues the conversation.)
CUL	See you later.
FOO?	A greeting, also meaning R U THERE? Often used in the case of unexpected links, meaning also "Sorry if I butted in" (linker) or "What's up?" (linkee).
FYI	For your information...
GA	Go ahead (used when two people have tried to type simultaneously; this cedes the right to type to the other).
HELLOP	A greeting, also meaning R U THERE? (An instance of the "-P" convention.)
MtFBWY	May the Force be with you. (From Star Wars.)
NIL	No (see the main entry for NIL).
OBTW	Oh, by the way...
R U THERE?	Are you there?
SEC	Wait a second (sometimes written SEC...).
T	Yes (see the main entry for T).
TNX	Thanks.
TNX 1.0E6	Thanks a million (humorous).
<double CRLF>	When the typing party has finished, he types two CRLF's to signal that he is done; this leaves a

blank line between individual "speeches" in the conversation, making it easier to re-read the preceding text.

<name>: When three or more terminals are linked, each speech is preceded by the typist's login name and a colon (or a hyphen) to indicate who is typing. The login name often is shortened to a unique prefix (possibly a single letter) during a very long conversation.

/\/\/\ The equivalent of a giggle.

At Stanford, where the link feature is implemented by "talk loops", the term TALK MODE is used in place of COM MODE. Most of the above "sub-jargon" is used at both Stanford and MIT.

CONNECTOR CONSPIRACY [probably came into prominence with the appearance of the KL-10, none of whose connectors match anything else] n. The tendency of manufacturers (or, by extension, programmers or purveyors of anything) to come up with new products which don't fit together with the old stuff, thereby making you buy either all new stuff or expensive interface devices.

CONS [from LISP] 1. v. To add a new element to a list. 2. CONS UP: v. To synthesize from smaller pieces: "to cons up an example".

CRASH 1. n. A sudden, usually drastic failure. Most often said of the system (q.v., definition #1), sometimes of magnetic disk drives. "Three lusers lost their files in last night's disk crash." A disk crash which entails the read/write heads dropping onto the surface of the disks and scraping off the oxide may also be referred to as a "head crash". 2. v. To fail suddenly. "Has the system just crashed?" Also used transitively to indicate the cause of the crash (usually a person or a program, or both). "Those idiots playing spacewar crashed the system." Sometimes said of people. See GRONK OUT.

CRETIN 1. n. Congenital loser (q.v.). 2. CRETINOUS: adj. See BLETCHEROUS and BAGBITING. Usage: somewhat ad hominem.

CRLF (cur'lif, sometimes crul'lif) n. A carriage return (CR) followed by a line feed (LF). See TERPRI.

CROCK [probably from "layman" slang, which in turn may be derived from "crock of shit"] n. An awkward feature or programming technique that ought to be made cleaner. Example: Using small integers to represent error codes without the program interpreting them to the user is a crock. Also, a technique that works acceptably but which is quite prone to failure if disturbed in the least, for example depending on the machine opcodes having particular bit patterns so that you can use instructions as data words too; a tightly woven, almost completely unmodifiable structure.

CRUFTY [from "cruddy"] adj. 1. Poorly built, possibly overly complex. "This is standard old crufty DEC software". Hence CRUFT, n. shoddy construction. 2. Unpleasant, especially to the touch, often with encrusted junk. Like spilled coffee smeared with peanut butter and catsup. Hence CRUFT, n. disgusting mess. 3. Generally unpleasant. CRUFTY or CRUFTIE n. A small crufty object (see FROB); often one which doesn't fit well into the scheme of things. "A LISP property list is a good place to store crufties (or, random cruft)." [Note: Does CRUFT have anything to do with the Cruft Lab at Harvard? I don't know, though I was a Harvard student. - GLS]

CRUNCH v. 1. To process, usually in a time-consuming or complicated way. Connotes an essentially trivial operation which is nonetheless painful to perform. The pain may be due to the triviality being imbedded in a loop from 1 to 1000000000. "FORTRAN programs do mostly number crunching." 2. To reduce the size of a file by a complicated scheme that produces bit configurations completely unrelated to the original data, such as by a Huffman code. (The file ends up looking like a paper document would if somebody crunched the paper into a wad.) Since such compression usually takes more computations than simpler methods such as counting repeated characters (such as spaces) the term is doubly appropriate. (This meaning is usually used in the construction "file crunch(ing)" to distinguish it from "number crunch(ing)".) 3. n. The character "#". Usage: used at Xerox and CMU, among other places. Other names for "#" include SHARP, NUMBER, HASH, PIG-PEN, POUND-SIGN, and MESH. GLS adds: I recall reading somewhere that most of these are names for the # symbol IN CONTEXT. The name for the sign itself is "octothorp".

CTY (city) n. The terminal physically associated with a computer's operating console.

CUSPY [from the DEC acronym CUSP, for Commonly Used System Program, i.e., a utility program used by many people] (WPI) adj. 1. (of a program) Well-written. 2. Functionally excellent. A program which performs well and interfaces well to users is cuspy. See RUDE.

DAEMON (day'mun, dee'mun) [archaic form of "demon", which has slightly different connotations (q.v.)] n. A program which is not invoked explicitly, but which lays dormant waiting for some condition(s) to occur. The idea is that the perpetrator of the condition need not be aware that a daemon is lurking (though often a program will commit an action only because it knows that it will implicitly invoke a daemon). For example, writing a file on the lpt spooler's directory will invoke the spooling daemon, which prints the file. The advantage is that programs which want (in this example) files printed need not compete for access to the lpt. They simply enter their implicit requests and let the daemon decide what to do with them. Daemons are usually spawned automatically by the system, and may either live forever or be regenerated at intervals. Usage: DAEMON and DEMON (q.v.) are often used interchangeably, but seem to have distinct connotations. DAEMON was introduced to computing by CTSS people (who pronounced it dee'mon) and used it to refer to what is now called a DRAGON or PHANTOM (q.v.). The meaning and pronunciation have drifted, and we think this glossary reflects current usage.

DAY MODE See PHASE (of people).

DEADLOCK n. A situation wherein two or more processes are unable to proceed because each is waiting for another to do something. A common example is a program communicating to a PTY or STY, which may find itself waiting for output from the PTY/STY before sending anything more to it, while the PTY/STY is similarly waiting for more input from the controlling program before outputting anything. (This particular flavor of deadlock is called "starvation". Another common flavor is "constipation", where each process is trying to send stuff to the other, but all buffers are full because nobody is reading anything.) See DEADLY EMBRACE.

DEADLY EMBRACE n. Same as DEADLOCK (q.v.), though usually used only when exactly two processes are involved. DEADLY EMBRACE is the more popular term in Europe; DEADLOCK in the United States.

DEMENTED adj. Yet another term of disgust used to describe a program. The connotation in this case is that the program works as designed, but the design is bad. For example, a program that generates large numbers of meaningless error messages implying it is on the point of imminent collapse.

DEMON (dee'mun) n. A portion of a program which is not invoked explicitly, but which lays dormant waiting for some condition(s) to occur. See DAEMON. The distinction is that demons are usually processes within a program, while daemons are usually programs running on an operating system. Demons are particularly common in AI programs. For example, a knowledge manipulation program might implement inference rules as demons. Whenever a new piece of knowledge was added, various demons would activate (which demons depends on the particular piece of data) and would create additional pieces of knowledge by applying their respective inference rules to the original piece. These new pieces could in turn activate more demons as the inferences filtered down through chains of logic. Meanwhile the main program could continue with whatever its primary task was.

DIABLO (dee-ah'blow) [from the Diablo printer] 1. n. Any letter-quality printing device. 2. v. To produce letter-quality output from such a device.

DIDDLE v. To work with in a not particularly serious manner. "I diddled with a copy of ADVENT so it didn't double-space all the time." "Let's diddle this piece of code and see if the problem goes away." See TWEAK and TWIDDLE.

DIKE [from "diagonal cutters"] v. To remove a module or disable it. "When in doubt, dike it out."

DMP (dump) See BIN.

DO PROTOCOL [from network protocol programming] v. To perform an interaction with somebody or something that follows a clearly defined procedure. For example, "Let's do protocol with the check" at a restaurant means to ask the waitress for the check, calculate the tip and everybody's share, generate change as necessary, and pay the bill.

DOWN 1. adj. Not working. "The up escalator is down." 2. TAKE DOWN, BRING DOWN: v. To deactivate, usually for repair work. See UP.

DPB (duh-pib') [from the PDP-10 instruction set] v. To plop something down in the middle.

DRAGON n. (MIT) A program similar to a "daemon" (q.v.), except that it is not invoked at all, but is instead used by the system to perform various secondary tasks. A typical example would be an accounting program, which keeps track of who is logged in, accumulates load-average statistics, etc. At MIT, all free TV's display a list of people logged in, where they are, what they're running, etc. along with some random picture (such as a unicorn, Snoopy, or the Enterprise) which is generated by the "NAME DRAGON". See PHANTOM.

DWIM [Do What I Mean] 1. adj. Able to guess, sometimes even correctly, what result was intended when provided with bogus input. Often suggested in jest as a desired feature for a complex program. A related term, more often seen as a verb, is DTRT (Do The Right Thing). 2. n. The INTERLISP function that attempts to accomplish this feat by correcting many of the more common errors. See HAIRY.

ENGLISH n. The source code for a program, which may be in any language, as opposed to BINARY. Usage: slightly obsolete, used mostly by old-time hackers, though recognizable in context. At MIT, directory SYSENG is where the "English" for system programs is kept, and SYSBIN, the binaries. SAIL has many such directories, but the canonical one is [CSP,SYS].

EPSILON [from standard mathematical notation for a small quantity] 1. n. A small quantity of anything. "The cost is epsilon." 2. adj. Very small, negligible; less than marginal (q.v.). "We can get this feature for epsilon cost." 3. WITHIN EPSILON OF: Close enough to be indistinguishable for all practical purposes.

EXCH (ex'chuh, ekstch) [from the PDP-10 instruction set] v. To exchange two things, each for the other.

EXCL (eks'cul) n. Abbreviation for "exclamation point". See BANG, SHRIEK, WOW.

EXE (ex'ee) See BIN.

FAULTY adj. Same denotation as "bagbiting", "bletcherous", "losing", q.v., but the connotation is much milder.

FEATURE n. 1. A surprising property of a program. Occasionally documented. To call a property a feature sometimes means the author of the program did not consider the particular case, and the program makes an unexpected, although not strictly speaking an incorrect response. See BUG. "That's not a bug, that's a feature!" A bug can be changed to a feature by documenting it. 2. A well-known and beloved property; a facility. Sometimes features are planned, but are called crocks by others. An approximately correct spectrum:

(These terms are all used to describe programs or portions thereof, except for the first two, which are included for completeness.)

```
     CRASH  STOPPAGE  BUG  SCREW  LOSS  MISFEATURE
                 CROCK  KLUGE  HACK  WIN  FEATURE  PERFECTION
```

(The last is never actually attained.)

FEEP 1. n. The soft bell of a display terminal (except for a VT-52!); a beep. 2. v. To cause the display to make a feep sound. TTY's do not have feeps. Alternate forms: BEEP, BLEEP, or just about anything suitably onomatopoeic. The term BREEDLE is sometimes heard at SAIL, where the terminal bleepers are not particularly "soft" (they sound more like the musical equivalent of sticking out one's tongue). The "feeper" on a VT-52 has been compared to the sound of a '52 Chevy stripping its gears.

FENCEPOST ERROR n. The discrete equivalent of a boundary condition. Often exhibited in programs by iterative loops. From the following problem: "If you build a fence 100 feet long with posts ten feet apart, how many posts do you need?" (Either 9 or 11 is a better answer than the obvious 10.)

FINE (WPI) adj. Good, but not good enough to be CUSPY. [The word FINE is used elsewhere, of course, but without the implicit comparison to the higher level implied by CUSPY.]

FLAG DAY [from a bit of Multics history involving a change in the ASCII character set originally scheduled for June 14, 1966] n. A software change which is neither forward nor backward compatible, and which is costly to make and costly to revert. "Can we install that without causing a flag day for all users?"

FLAKEY adj. Subject to frequent lossages. See LOSSAGE.

FLAME v. To speak incessantly and/or rabidly on some relatively uninteresting subject or with a patently ridiculous attitude. FLAME ON: v. To continue to flame. See RAVE.

FLAP v. To unload a DECtape (so it goes flap, flap, flap...). Old hackers at MIT tell of the days when the disk was device 0 and microtapes were 1, 2,... and attempting to flap device 0 would instead start a motor banging inside a cabinet near the disk!

FLAVOR n. 1. Variety, type, kind. "DDT commands come in two flavors." See VANILLA. 2. The attribute of causing something to be FLAVORFUL. "This convention yields additional flavor by allowing one to..." 3. On the LispMachine, an object-oriented programming system ("flavors"); each class of object is a flavor.

FLAVORFUL adj. Aesthetically pleasing. See RANDOM and LOSING for antonyms. See also the entry for TASTE.

FLUSH v. 1. To delete something, usually superfluous. "All that nonsense has been flushed." Standard ITS terminology for aborting an output operation. 2. To leave at the end of a day's work (as opposed to leaving for a meal). "I'm going to flush now." "Time to flush." 3. To exclude someone from an activity.

FOO 1. [from Yiddish "feh" or the Anglo-Saxon "fooey!"] interj. Term of disgust. 2. [from FUBAR (Fucked Up Beyond All Recognition), from WWII, often seen as FOOBAR] Name used for temporary programs, or samples of three-letter names. Other similar words are BAR, BAZ (Stanford corruption of BAR), and rarely RAG. These have been used in Pogo as well. 3. Used very generally as a sample name for absolutely anything. The old 'Smokey Stover' comic strips often included the word FOO, in particular on license plates of cars. MOBY FOO: See MOBY.

FRIED adj. 1. Non-working due to hardware failure; burnt out. 2. Of people, exhausted. Said particularly of those who continue to work in such a state. Often used as an explanation or excuse. "Yeah, I know that fix destroyed the file system, but I was fried when I put it in."

FROB 1. n. (MIT) The official Tech Model Railroad Club definition is "FROB = protruding arm or trunnion", and by metaphoric extension any somewhat small thing. See FROBNITZ. 2. v. Abbreviated form of FROBNICATE.

FROBNICATE v. To manipulate or adjust, to tweak. Derived from FROBNITZ (q.v.). Usually abbreviated to FROB. Thus one has the saying "to frob a frob". See TWEAK and TWIDDLE. Usage: FROB, TWIDDLE, and TWEAK sometimes connote points along a continuum.

FROB connotes aimless manipulation; TWIDDLE connotes gross manipulation, often a coarse search for a proper setting; TWEAK connotes fine-tuning. If someone is turning a knob on an oscilloscope, then if he's carefully adjusting it he is probably tweaking it; if he is just turning it but looking at the screen he is probably twiddling it; but if he's just doing it because turning a knob is fun, he's frobbing it.

FROBNITZ, pl. FROBNITZEM (frob'nitsm) n. An unspecified physical object, a widget. Also refers to electronic black boxes. This rare form is usually abbreviated to FROTZ, or more commonly to FROB. Also used are FROBNULE, FROBULE, and FROBNODULE. Starting perhaps in 1979, FROBBOZ (fruh-bahz'), pl. FROBBOTZIM, has also become very popular, largely due to its exposure via the Adventure spin-off called Zork (Dungeon). These can also be applied to non-physical objects, such as data structures.

FROG (variant: PHROG) 1. interj. Term of disgust (we seem to have a lot of them). 2. Used as a name for just about anything. See FOO. 3. n. Of things, a crock. Of people, somewhere inbetween a turkey and a toad. 4. Jake Brown (FRG@SAIL). 5. FROGGY: adj. Similar to BAGBITING (q.v.), but milder. "This froggy program is taking forever to run!"

FROTZ 1. n. See FROBNITZ. 2. MUMBLE FROTZ: An interjection of very mild disgust.

FRY v. 1. To fail. Said especially of smoke-producing hardware failures. 2. More generally, to become non-working. Usage: never said of software, only of hardware and humans. See FRIED.

FTP (spelled out, NOT pronounced "fittip") 1. n. The File Transfer Protocol for transmitting files between systems on the ARPAnet. 2. v. To transfer a file using the File Transfer Program. "Lemme get this copy of Wuthering Heights FTP'd from SAIL."

FUDGE 1. v. To perform in an incomplete but marginally acceptable way, particularly with respect to the writing of a program. "I didn't feel like going through that pain and suffering, so I fudged it." 2. n. The resulting code.

FUDGE FACTOR n. A value or parameter that is varied in an ad hoc way to produce the desired result. The terms "tolerance" and "slop" are also used, though these usually indicate a one-sided leeway, such as a buffer which is made larger than necessary because one isn't sure exactly how large it needs to be, and it is better to waste a little space than to lose completely for not having enough. A fudge factor, on the other hand, can often be tweaked in more than one direction. An example might be the coefficients of an equation, where the coefficients are varied in an attempt to make the equation fit certain criteria.

GABRIEL [for Dick Gabriel, SAIL volleyball fanatic] n. An unnecessary (in the opinion of the opponent) stalling tactic, e.g., tying one's shoelaces or hair repeatedly, asking the time, etc. Also used to refer to the perpetrator of such tactics. Also, "pulling a Gabriel", "Gabriel mode".

GARBAGE COLLECT v., GARBAGE COLLECTION n. See GC.

GARPLY n. (Stanford) Another meta-word popular among SAIL hackers.

GAS [as in "gas chamber"] interj. 1. A term of disgust and hatred, implying that gas should be dispensed in generous quantities, thereby exterminating the source of irritation. "Some loser just reloaded the system for no reason! Gas!" 2. A term suggesting that someone or something ought to be flushed out of mercy. "The system's wedging every few minutes. Gas!" 3. v. FLUSH (q.v.). "You should gas that old crufty software." 4. GASEOUS adj. Deserving of being gassed. Usage: primarily used by Geoff Goodfellow at SRI, but spreading.

GC [from LISP terminology] 1. v. To clean up and throw away useless things. "I think I'll GC the top of my desk today." 2. v. To recycle, reclaim, or put to another use. 3. n. An instantiation of the GC process.

GEDANKEN [from Einstein's term "gedanken-experimenten", such as the standard proof that E=mc^2] adj. An AI project which is written up in grand detail without ever being implemented to any great extent. Usually perpetrated by people who aren't very good hackers or find programming distasteful or are just in a hurry. A gedanken thesis is usually marked by an obvious lack of intuition about what is programmable and what is not and about what does and does not constitute a clear specification of a program-related concept such as an algorithm.

GLASS TTY n. A terminal which has a display screen but which, because of hardware or software limitations, behaves like a teletype or other printing terminal. An example is the ADM-3 (without cursor control). A glass tty can't do neat display hacks, and you can't save the output either.

GLITCH [from the Yiddish "glitshen", to slide] 1. n. A sudden interruption in electric service, sanity, or program function. Sometimes recoverable. 2. v. To commit a glitch. See GRITCH. 3. v. (Stanford) To scroll a display screen.

GLORK 1. interj. Term of mild surprise, usually tinged with outrage, as when one attempts to save the results of two hours of editing and finds that the system has just crashed. 2. Used as a name for just about anything. See FOO. 3. v. Similar to GLITCH (q.v.), but usually used reflexively. "My program just glorked itself."

GOBBLE v. To consume or to obtain. GOBBLE UP tends to imply "consume", while GOBBLE DOWN tends to imply "obtain". "The output spy gobbles characters out of a TTY output buffer." "I guess I'll gobble down a copy of the documentation tomorrow." See SNARF.

GORP (CMU) [perhaps from the generic term for dried hiker's food, stemming from the acronym "Good Old Raisins and Peanuts"] Another metasyntactic variable, like FOO and BAR.

GRIND v. 1. (primarily MIT) To format code, especially LISP code, by indenting lines so that it looks pretty. Hence, PRETTY PRINT, the generic term for such operations. 2. To run seemingly interminably, performing some tedious and inherently useless task. Similar to CRUNCH.

GRITCH 1. n. A complaint (often caused by a GLITCH (q.v.)). 2. v. To complain. Often verb-doubled: "Gritch gritch". 3. Glitch.

GROK [from the novel "Stranger in a Strange Land", by Robert Heinlein, where it is a Martian word meaning roughly "to be one with"] v. To understand, usually in a global sense.

GRONK [popularized by the cartoon strip "B.C." by Johnny Hart, but the word apparently predates that] v. 1. To clear the state of a wedged device and restart it. More severe than "to frob" (q.v.). 2. To break. "The teletype scanner was gronked, so we took the system down." 3. GRONKED: adj. Of people, the condition of feeling very tired or sick. 4. GRONK OUT: v. To cease functioning. Of people, to go home and go to sleep. "I guess I'll gronk out now; see you all tomorrow."

GROVEL v. To work interminably and without apparent progress. Often used with "over". "The compiler grovelled over my code." Compare GRIND and CRUNCH. Emphatic form: GROVEL OBSCENELY.

GRUNGY adj. Incredibly dirty or grubby. Anything which has been washed within the last year is not really grungy. Also used metaphorically; hence some programs (especially crocks) can be described as grungy.

GUBBISH [a portmanteau of "garbage" and "rubbish"?] n. Garbage; crap; nonsense. "What is all this gubbish?"

HACK n. 1. Originally a quick job that produces what is needed, but not well. 2. The result of that job. 3. NEAT HACK: A clever technique. Also, a brilliant practical joke, where neatness is correlated with cleverness, harmlessness, and surprise value. Example: the Caltech Rose Bowl card display switch circa 1961. 4. REAL HACK: A crock (occasionally affectionate).

changing the JFCL to a PUSHJ one can insert a debugging routine at that point.

HUMONGOUS, HUMUNGOUS See HUNGUS.

HUNGUS (hung'ghis) [perhaps related to current slang "humongous"; which one came first (if either) is unclear] adj. Large, unwieldy, usually unmanageable. "TCP is a hungus piece of code." "This is a hungus set of modifications."

IMPCOM See TELNET.

INFINITE adj. Consisting of a large number of objects; extreme. Used very loosely as in: "This program produces infinite garbage."

IRP (erp) [from the MIDAS pseudo-op which generates a block of code repeatedly, substituting in various places the car and/or cdr of the list(s) supplied at the IRP] v. To perform a series of tasks repeatedly with a minor substitution each time through. "I guess I'll IRP over these homework papers so I can give them some random grade for this semester."

JFCL (djif'kl or dja-fik'l) [based on the PDP-10 instruction that acts as a fast no-op] v. To cancel or annul something. "Why don't you jfcl that out?" [The licence plate on Geoff Goodfellow's BMW is JFCL.]

JIFFY n. 1. Interval of CPU time, commonly 1/60 second or 1 millisecond. 2. Indeterminate time from a few seconds to forever. "I'll do it in a jiffy" means certainly not now and possibly never.

JOCK n. Programmer who is characterized by large and somewhat brute force programs. The term is particularly well-suited for systems programmers.

J. RANDOM See RANDOM.

JRST (jerst) [based on the PDP-10 jump instruction] v. To suddenly change subjects. Usage: rather rare. "Jack be nimble, Jack be quick; Jack jrst over the candle stick."

JSYS (jay'sis), pl. JSI (jay'sigh) [Jump to SYStem] See UUO.

KLUGE (kloodj) alt. KLUDGE [from the German "kluge", clever] n. 1. A Rube Goldberg device in hardware or software. 2. A clever programming trick intended to solve a particular nasty case in an efficient, if not clear, manner. Often used to repair bugs. Often verges on being a crock. 3. Something that works for the wrong reason. 4. v. To insert a kluge into a program. "I've kluged this routine to get around that weird bug, but there's probably a better way." Also KLUGE UP. 5. KLUGE AROUND: To avoid by inserting a kluge. 6. (WPI) A feature which is implemented in a RUDE manner.

LDB (lid'dib) [from the PDP-10 instruction set] v. To extract from the middle.

LIFE n. A cellular-automata game invented by John Horton Conway, and first introduced publicly by Martin Gardner (Scientific American, October 1970).

LINE FEED (standard ASCII terminology) 1. v. To feed the paper through a terminal by one line (in order to print on the next line). 2. n. The "character" which causes the terminal to perform this action.

LINE STARVE (MIT) Inverse of LINE FEED.

LOGICAL [from the technical term "logical device", wherein a physical device is referred to by an arbitrary name] adj. Understood to have a meaning not necessarily corresponding to reality. E.g., if a person who has long held a certain post (e.g., Les Earnest at SAIL) left and was replaced, the replacement would for a while be known as the "logical Les Earnest". The word VIRTUAL is also used. At SAIL, "logical" compass directions denote a coordinate system in which "logical north" is toward San Francisco, "logical west" is toward the ocean, etc., even though logical north varies between physical (true) north near SF and physical west near San Jose.

(The best rule of thumb here is that El Camino Real by definition always runs logical north-and-south.)

LOSE [from MIT jargon] v. 1. To fail. A program loses when it encounters an exceptional condition. 2. To be exceptionally unaesthetic. 3. Of people, to be obnoxious or unusually stupid (as opposed to ignorant). 4. DESERVE TO LOSE: v. Said of someone who willfully does the wrong thing; humorously, if one uses a feature known to be marginal. What is meant is that one deserves the consequences of one's losing actions. "Boy, anyone who tries to use MULTICS deserves to lose!"
LOSE LOSE - a reply or comment on a situation.

LOSER n. An unexpectedly bad situation, program, programmer, or person. Especially "real loser".

LOSS n. Something which loses. WHAT A (MOBY) LOSS!: interjection.

LOSSAGE n. The result of a bug or malfunction.

LPT (lip'it) n. Line printer, of course.

LUSER See USER.

MACROTAPE n. An industry standard reel of tape, as opposed to a MICROTAPE.

MAGIC adj. 1. As yet unexplained, or too complicated to explain. (Arthur C. Clarke once said that magic was as-yet-not-understood science.) "TTY echoing is controlled by a large number of magic bits." "This routine magically computes the parity of an eight-bit byte in three instructions." 2. (Stanford) A feature not generally publicized which allows something otherwise impossible, or a feature formerly in that category but now unveiled. Example: The keyboard commands which override the screen-hiding features.

MARGINAL adj. 1. Extremely small. "A marginal increase in core can decrease GC time drastically." See EPSILON. 2. Of extremely small merit. "This proposed new feature seems rather marginal to me." 3. Of extremely small probability of winning. "The power supply was rather marginal anyway; no wonder it crapped out." 4. MARGINALLY: adv. Slightly. "The ravs here are only marginally better than at Small Eating Place."

MICROTAPE n. Occasionally used to mean a DECtape, as opposed to a MACROTAPE.

MISFEATURE n. A feature which eventually screws someone, possibly because it is not adequate for a new situation which has evolved. It is not the same as a bug because fixing it involves a gross philosophical change to the structure of the system involved. Often a former feature becomes a misfeature because a tradeoff was made whose parameters subsequently changed (possibly only in the judgment of the implementors). "Well, yeah, it's kind of a misfeature that file names are limited to six characters, but we're stuck with it for now."

MOBY [seems to have been in use among model railroad fans years ago. Entered the world of AI with the Fabritek 256K moby memory of MIT-AI. Derived from Melville's "Moby Dick" (some say from "Moby Pickle").] 1. adj. Large, immense, or complex. "A moby frob." 2. n. The maximum address space of a machine, hence 3. n. 256K words, the size of a PDP-10 moby. (The maximum address space means the maximum normally addressable space, as opposed to the amount of physical memory a machine can have. Thus the MIT PDP-10s each have two mobies, usually referred to as the "low moby" (0-777777) and "high moby" (1000000-1777777), or as "moby 0" and "moby 1". MIT-AI has four mobies of address space: moby 2 is the PDP-6 memory, and moby 3 the PDP-11 interface.) In this sense "moby" is often used as a generic unit of either address space (18. bits' worth) or of memory (about a megabyte, or 9/8 megabyte (if one accounts for difference between 32.- and 36.-bit words), or 5/4 megacharacters). 4. A title of address (never of third-person reference), usually used to show admiration, respect, and/or friendliness to a competent hacker. "So, moby Knight, how's the CONS machine doing?" 5. adj. In backgammon, doubles on the dice, as in "moby sixes",

"moby ones", etc.
MOBY FOO, MOBY WIN, MOBY LOSS: standard emphatic forms.
FOBY MOO: a spoonerism due to Greenblatt.

MODE n. A general state, usually used with an adjective describing the state. "No time to hack; I'm in thesis mode." Usage: in its jargon sense, MODE is most often said of people, though it is sometimes applied to programs and inanimate objects. "If you're on a TTY, E will switch to non-display mode." In particular, see DAY MODE, NIGHT MODE, and YOYO MODE; also COM MODE, TALK MODE, and GABRIEL MODE.

MODULO prep. Except for. From mathematical terminology: one can consider saying that 4=22 "except for the 9's" (4=22 mod 9). "Well, LISP seems to work okay now, modulo that GC bug."

MOON n. 1. A celestial object whose phase is very important to hackers. See PHASE OF THE MOON. 2. Dave Moon (MOON@MC).

MUMBLAGE n. The topic of one's mumbling (see MUMBLE). "All that mumblage" is used like "all that stuff" when it is not quite clear what it is or how it works, or like "all that crap" when "mumble" is being used as an implicit replacement for obscenities.

MUMBLE interj. 1. Said when the correct response is either too complicated to enunciate or the speaker has not thought it out. Often prefaces a longer answer, or indicates a general reluctance to get into a big long discussion. "Well, mumble." 2. Sometimes used as an expression of disagreement. "I think we should buy it." "Mumble!" Common variant: MUMBLE FROTZ. 3. Yet another metasyntactic variable, like FOO.

MUNCH (often confused with "mung", q.v.) v. To transform information in a serial fashion, often requiring large amounts of computation. To trace down a data structure. Related to CRUNCH (q.v.), but connotes less pain.

MUNCHING SQUARES n. A display hack dating back to the PDP-1, which employs a trivial computation (involving XOR'ing of x-y display coordinates - see HAKMEM items 146-148) to produce an impressive display of moving, growing, and shrinking squares. The hack usually has a parameter (usually taken from toggle switches) which when well-chosen can produce amazing effects. Some of these, discovered recently on the LISP machine, have been christened MUNCHING TRIANGLES, MUNCHING W'S, and MUNCHING MAZES.

MUNG (variant: MUNGE) [recursive acronym for Mung Until No Good] v. 1. To make changes to a file, often large-scale, usually irrevocable. Occasionally accidental. See BLT. 2. To destroy, usually accidentally, occasionally maliciously. The system only mungs things maliciously.

N adj. 1. Some large and indeterminate number of objects; "There were N bugs in that crock!"; also used in its original sense of a variable name. 2. An arbitrarily large (and perhaps infinite) number. 3. A variable whose value is specified by the current context. "We'd like to order N wonton soups and a family dinner for N-1." 4. NTH: adj. The ordinal counterpart of N. "Now for the Nth and last time..." In the specific context "Nth-year grad student", N is generally assumed to be at least 4, and is usually 5 or more. See also 69.

NIGHT MODE See PHASE (of people).

NIL [from LISP terminology for "false"] No. Usage: used in reply to a question, particularly one asked using the "-P" convention. See T.

OBSCURE adj. Used in an exaggeration of its normal meaning, to imply a total lack of comprehensibility. "The reason for that last crash is obscure." "FIND's command syntax is obscure." MODERATELY OBSCURE implies that it could be figured out but probably isn't worth the trouble.

OPEN n. Abbreviation for "open (or left) parenthesis", used when necessary to eliminate oral ambiguity. To read aloud the LISP form (DEFUN FOO (X) (PLUS X 1)) one might say: "Open def-fun foo, open eks close, open, plus ekx one, close close." See CLOSE.

PARSE [from linguistic terminology] v. 1. To determine the syntactic structure of a sentence or other utterance (close to the standard English meaning). Example: "That was the one I saw you." "I can't parse that." 2. More generally, to understand or comprehend. "It's very simple; you just kretch the glims and then aos the zotz." "I can't parse that." 3. Of fish, to have to remove the bones yourself (usually at a Chinese restaurant). "I object to parsing fish" means "I don't want to get a whole fish, but a sliced one is okay." A "parsed fish" has been deboned. There is some controversy over whether "unparsed" should mean "bony", or also mean "deboned".

PATCH 1. n. A temporary addition to a piece of code, usually as a quick-and-dirty remedy to an existing bug or misfeature. A patch may or may not work, and may or may not eventually be incorporated permanently into the program. 2. v. To insert a patch into a piece of code.

PDL (piddle or puddle) [acronym for Push Down List] n. 1. A LIFO queue (stack); more loosely, any priority queue; even more loosely, any queue. A person's pdl is the set of things he has to do in the future. One speaks of the next project to be attacked as having risen to the top of the pdl. "I'm afraid I've got real work to do, so this'll have to be pushed way down on my pdl." See PUSH and POP. 2. Dave Lebling (PDL@DM).

PESSIMAL [Latin-based antonym for "optimal"] adj. Maximally bad. "This is a pessimal situation."

PESSIMIZING COMPILER n. A compiler that produces object code that is worse than the straightforward or obvious translation.

PHANTOM n. (Stanford) The SAIL equivalent of a DRAGON (q.v.). Typical phantoms include the accounting program, the news-wire monitor, and the lpt and xgp spoolers.

PHASE (of people) 1. n. The phase of one's waking-sleeping schedule with respect to the standard 24-hour cycle. This is a useful concept among people who often work at night according to no fixed schedule. It is not uncommon to change one's phase by as much as six hours/day on a regular basis. "What's your phase?" "I've been getting in about 8 PM lately, but I'm going to work around to the day schedule by Friday." A person who is roughly 12 hours out of phase is sometimes said to be in "night mode". (The term "day mode" is also used, but less frequently.) 2. CHANGE PHASE THE HARD WAY: To stay awake for a very long time in order to get into a different phase. 3. CHANGE PHASE THE EASY WAY: To stay asleep etc.

PHASE OF THE MOON n. Used humorously as a random parameter on which something is said to depend. Sometimes implies unreliability of whatever is dependent, or that reliability seems to be dependent on conditions nobody has been able to determine. "This feature depends on having the channel open in mumble mode, having the foo switch set, and on the phase of the moon."

PLUGH [from the Adventure game] v. See XYZZY.

POM n. Phase of the moon (q.v.). Usage: usually used in the phrase "POM dependent" which means flakey (q.v.).

POP [based on the stack operation that removes the top of a stack, and the fact that procedure return addresses are saved on the stack] dialect: POPJ (pop-jay), based on the PDP-10 procedure return instruction. v. To return from a digression. By verb doubling, "Popj, popj" means roughly, "Now let's see, where were we?"

PPN (pip'in) [DEC terminology, short for Project-Programmer Number] n. 1. A combination 'project' (directory name) and programmer name, used to identify a specific directory belonging to that user. For instance, "FOO,BAR" would be the FOO directory for user BAR. Since the name is restricted to three letters, the programmer name is usually the person's initials, though sometimes it is a nickname or other special sequence. (Standard DEC setup is to have two octal numbers instead of characters; hence the original acronym.) 2.

Often used loosely to refer to the programmer name alone. "I want to send you some mail; what's your ppn?" Usage: not used at MIT, since ITS does not use ppn's. The equivalent terms would be UNAME and SNAME, depending on context, but these are not used except in their technical senses.

PROTOCOL See DO PROTOCOL.

PSEUDOPRIME n. A backgammon prime (six consecutive occupied points) with one point missing.

PTY (pity) n. Pseudo TTY, a simulated TTY used to run a job under the supervision of another job.
PTYJOB (pity-job) n. The job being run on the PTY. Also a common general-purpose program for creating and using PTYs.
This is DEC and SAIL terminology; the MIT equivalent is STY.

PUNT [from the punch line of an old joke: "Drop back 15 yards and punt"] v. To give up, typically without any intention of retrying.

PUSH [based on the stack operation that puts the current information on a stack, and the fact that procedure call addresses are saved on the stack] dialect: PUSHJ (push-jay), based on the PDP-10 procedure call instruction. v. To enter upon a digression, to save the current discussion for later.

QUES (kwess) 1. n. The question mark character ("?"). 2. interj. What? Also QUES QUES? See WALL.

QUUX [invented by Steele. Mythically, from the Latin semi-deponent verb QUUXO, QUUXARE, QUUXANDUM IRI; noun form variously QUUX (plural QUUCES, Anglicized to QUUXES) and QUUXU (genitive plural is QUUXUUM, four U's in seven letters).] 1. Originally, a meta-word like FOO and FOOBAR. Invented by Guy Steele for precisely this purpose when he was young and naive and not yet interacting with the real computing community. Many people invent such words; this one seems simply to have been lucky enough to have spread a little. 2. interj. See FOO; however, denotes very little disgust, and is uttered mostly for the sake of the sound of it. 3. n. Refers to one of four people who went to Boston Latin School and eventually to MIT:

THE GREAT QUUX:	Guy L. Steele Jr.
THE LESSER QUUX:	David J. Littleboy
THE MEDIOCRE QUUX:	Alan P. Swide
THE MICRO QUUX:	Sam Lewis

(This taxonomy is said to be similarly applied to three Frankston brothers at MIT.) QUUX, without qualification, usually refers to The Great Quux, who is somewhat infamous for light verse and for the "Crunchly" cartoons. 4. QUUXY: adj. Of or pertaining to a QUUX.

RANDOM adj. 1. Unpredictable (closest to mathematical definition); weird. "The system's been behaving pretty randomly." 2. Assorted; undistinguished. "Who was at the conference?" "Just a bunch of random business types." 3. Frivolous; unproductive; undirected (pejorative). "He's just a random loser." 4. Incoherent or inelegant; not well organized. "The program has a random set of misfeatures." "That's a random name for that function." "Well, all the names were chosen pretty randomly." 5. Gratuitously wrong, i.e., poorly done and for no good apparent reason. For example, a program that handles file name defaulting in a particularly useless way, or a routine that could easily have been coded using only three ac's, but randomly uses seven for assorted non-overlapping purposes, so that no one else can invoke it without first saving four extra ac's. 6. In no particular order, though deterministic. "The I/O channels are in a pool, and when a file is opened one is chosen randomly." n. 7. A random hacker; used particularly of high school students who soak up computer time and generally get in the way. 8. (occasional MIT usage) One who lives at Random Hall.
J. RANDOM is often prefixed to a noun to make a "name" out of it (by comparison to common names such as "J. Fred Muggs"). The most common uses are "J. Random Loser" and "J. Random Nurd" ("Should J. Random Loser be allowed to gun down other people?"), but it can be used just as an elaborate version of RANDOM in any sense.
[The word RANDOM doesn't really have 69 different meanings. In fact, RANDOM has only one meaning, an extremely subtle and profound

one which defies articulation. Which connotation a given RANDOM-token has depends in similarly profound ways on the context. Similar comments apply to a couple other hacker jargon items, most notably HACK. - Agre]

RANDOMNESS n. An unexplainable misfeature; gratuitous inelegance. Also, a hack or crock which depends on a complex combination of coincidences (or rather, the combination upon which the crock depends). "This hack can output characters 40-57 by putting the character in the accumulator field of an XCT and then extracting 6 bits -- the low two bits of the XCT opcode are the right thing." "What randomness!"

RAPE v. To (metaphorically) screw someone or something, violently. Usage: often used in describing file-system damage. "So-and-so was running a program that did absolute disk I/O and ended up raping the master directory."

RAVE (WPI) v. 1. To persist in discussing a specific subject. 2. To speak authoritatively on a subject about which one knows very little. 3. To complain to a person who is not in a position to correct the difficulty. 4. To purposely annoy another person verbally. 5. To evangelize. See FLAME. Also used to describe a less negative form of blather, such as friendly bullshitting.

REAL USER n. 1. A commercial user. One who is paying "real" money for his computer usage. 2. A non-hacker. Someone using the system for an explicit purpose (research project, course, etc.). See USER.

REAL WORLD, THE n. 1. In programming, those institutions at which programming may be used in the same sentence as FORTRAN, COBOL, RPG, IBM, etc. 2. To programmers, the location of non-programmers and activities not related to programming. 3. A universe in which the standard dress is shirt and tie and in which a person's working hours are defined as 9 to 5. 4. The location of the status quo. 5. Anywhere outside a university. "Poor fellow, he's left MIT and gone into the real world." Used pejoratively by those not in residence there. In conversation, talking of someone who has entered the real world is not unlike talking about a deceased person.

REL See BIN.

RIGHT THING, THE n. That which is "obviously" the correct or appropriate thing to use, do, say, etc. Use of this term often implies that in fact reasonable people may disagree. "Never let your conscience keep you from doing the right thing!" "What's the right thing for LISP to do when it reads '(.)'?"

RUDE (WPI) adj. 1. (of a program) Badly written. 2. Functionally poor, e.g. a program which is very difficult to use because of gratuitously poor (random?) design decisions. See CUSPY.

SACRED adj. Reserved for the exclusive use of something (a metaphorical extension of the standard meaning). "Accumulator 7 is sacred to the UUO handler." Often means that anyone may look at the sacred object, but clobbering it will screw whatever it is sacred to.

SAGA (WPI) n. A cuspy but bogus raving story dealing with N random broken people.

SAV (save) See BIN.

SEMI 1. n. Abbreviation for "semicolon", when speaking. "Commands to GRIND are prefixed by semi-semi-star" means that the prefix is ";;*", not 1/4 of a star. 2. Prefix with words such as "immediately", as a qualifier. "When is the system coming up?" "Semi-immediately."

SERVER n. A kind of DAEMON which performs a service for the requester, which often runs on a computer other than the one on which the server runs.

SHIFT LEFT (RIGHT) LOGICAL [from any of various machines' instruction sets] 1. v. To move oneself to the left (right). To move out of the way. 2. imper. Get out of that (my) seat! Usage: often used without the "logical", or as "left shift" instead of "shift left". Sometimes heard as LSH (lish), from the PDP-10 instruction set.

SHR (share or shir) See BIN.

SHRIEK See EXCL. (Occasional CMU usage.)

69 adj. Large quantity. Usage: Exclusive to MIT-AI. "Go away, I have 69 things to do to DDT before worrying about fixing the bug in the phase of the moon output routine..."
[Note: Actually, any number less than 100 but large enough to have no obvious magic properties will be recognized as a "large number". There is no denying that "69" is the local favorite. I don't know whether its origins are related to the obscene interpretation, but I do know that 69 decimal = 105 octal, and 69 hexadecimal = 105 decimal, which is a nice property. - GLS]

SLOP n. 1. A one-sided fudge factor (q.v.). Often introduced to avoid the possibility of a fencepost error (q.v.). 2. (used by compiler freaks) The ratio of code generated by a compiler to hand-compiled code, minus 1; i.e., the space (or maybe time) you lose because you didn't do it yourself.

SLURP v. To read a large data file entirely into core before working on it. "This program slurps in a 1K-by-1K matrix and does an FFT."

SMART adj. Said of a program that does the right thing (q.v.) in a wide variety of complicated circumstances. There is a difference between calling a program smart and calling it intelligent; in particular, there do not exist any intelligent programs.

SMOKING CLOVER n. A psychedelic color munch due to Gosper.

SMOP [Simple (or Small) Matter of Programming] n. A piece of code, not yet written, whose anticipated length is significantly greater than its complexity. Usage: used to refer to a program that could obviously be written, but is not worth the trouble.

SNARF v. To grab, esp. a large document or file for the purpose of using it either with or without the author's permission. See BLT. Variant: SNARF (IT) DOWN. (At MIT on ITS, DDT has a command called :SNARF which grabs a job from another (inferior) DDT.)

SOFTWARE ROT n. Hypothetical disease the existence of which has been deduced from the observation that unused programs or features will stop working after sufficient time has passed, even if "nothing has changed". Also known as "bit decay".

SOFTWARILY adv. In a way pertaining to software. "The system is softwarily unreliable." The adjective "softwary" is NOT used. See HARDWARILY.

SOS 1. (ess-oh-ess) n. A losing editor, SON OF STOPGAP. 2. (sahss) v. Inverse of AOS, from the PDP-10 instruction set.

SPAZZ 1. v. To behave spastically or erratically; more often, to commit a single gross error. "Boy, is he spazzing!" 2. n. One who spazzes. "Boy, what a spazz!" 3. n. The result of spazzing. "Boy, what a spazz!"

SPLAT n. 1. Name used in many places (DEC, IBM, and others) for the ASCII star ("*") character. 2. (MIT) Name used by some people for the ASCII pound-sign ("#") character. 3. (Stanford) Name used by some people for the Stanford/ITS extended ASCII circle-x character. (This character is also called "circle-x", "blobby", and "frob", among other names.) 4. (Stanford) Name for the semi-mythical extended ASCII circle-plus character. 5. Canonical name for an output routine that outputs whatever the the local interpretation of splat is. Usage: nobody really agrees what character "splat" is, but the term is common.

SUPDUP v. To communicate with another ARPAnet host using the SUPDUP program, which is a SUPer-DUPer TELNET talking a special display protocol used mostly in talking to ITS sites. Sometimes abbreviated to SD.

STATE n. Condition, situation. "What's the state of NEWIO?" "It's winning away." "What's your state?" "I'm about to gronk out." As a special case, "What's the state of the world?" (or, more silly, "State-of-world-P?") means "What's new?" or "What's going on?"

STOPPAGE n. Extreme lossage (see LOSSAGE) resulting in something (usually vital) becoming completely unusable.

STY (pronounced "sty", not spelled out) n. A pseudo-teletype, which is a two-way pipeline with a job on one end and a fake keyboard-tty on the other. Also, a standard program which provides a pipeline from its controlling tty to a pseudo-teletype (and thence to another tty, thereby providing a "sub-tty").
This is MIT terminology; the SAIL and DEC equivalent is PTY.

SUPERPROGRAMMER n. See "wizard", "hacker". Usage: rare. (Becoming more common among IBM and Yourdon types.)

SWAPPED adj. From the use of secondary storage devices to implement virtual memory in computer systems. Something which is SWAPPED IN is available for immediate use in main memory, and otherwise is SWAPPED OUT. Often used metaphorically to refer to people's memories ("I read TECO ORDER every few months to keep the information swapped in.") or to their own availability ("I'll swap you in as soon as I finish looking at this other problem.").

SYSTEM n. 1. The supervisor program on the computer. 2. Any large-scale program. 3. Any method or algorithm. 4. The way things are usually done. Usage: a fairly ambiguous word. "You can't beat the system."
SYSTEM HACKER: one who hacks the system (in sense 1 only; for sense 2 one mentions the particular program: e.g., LISP HACKER)

T [from LISP terminology for "true"] 1. Yes. Usage: used in reply to a question, particularly one asked using the "-P" convention). See NIL. 2. See TIME T.

TALK MODE See COM MODE.

TASTE n. (primarily MIT-DMS) The quality in programs which tends to be inversely proportional to the number of features, hacks, and kluges programmed into it. Also, TASTY, TASTEFUL, TASTEFULNESS. "This feature comes in N tasty flavors." Although TASTEFUL and FLAVORFUL are essentially synonyms, TASTE and FLAVOR are not.

TECO (tee'koe) [acronym for Text Editor and COrrector] 1. n. A text editor developed at MIT, and modified by just about everybody. If all the dialects are included, TECO might well be the single most prolific editor in use. Noted for its powerful pseudo-programming features and its incredibly hairy syntax. 2. v. To edit using the TECO editor in one of its infinite forms; sometimes used to mean "to edit" even when not using TECO! Usage: rare at SAIL, where most people wouldn't touch TECO with a TENEX pole.
[Historical note: DEC grabbed an ancient version of MIT TECO many years ago when it was still a TTY-oriented editor. By now, TECO at MIT is highly display-oriented and is actually a language for writing editors, rather than an editor. Meanwhile, the outside world's various versions of TECO remain almost the same as the MIT version of ten years ago. DEC recently tried to discourage its use, but an underground movement of sorts kept it alive.]
[Since this note was written I found out that DEC tried to force their hackers by administrative decision to use a hacked up and generally lobotomized version of SOS instead of TECO, and they revolted. - MRC]

TELNET v. To communicate with another ARPAnet host using the TELNET protocol. TOPS-10 people use the word IMPCOM since that is the program name for them. Sometimes abbreviated to TN. "I usually TN over to SAIL just to read the AP News."

TENSE adj. Of programs, very clever and efficient. A tense piece of code often got that way because it was highly bummed, but sometimes it was just based on a great idea. A comment in a clever display routine by Mike Kazar: "This routine is so tense it will bring tears to your eyes. Much thanks to Craig Everhart and James Gosling for inspiring this hack attack." A tense programmer is one who produces tense code.

TERPRI (tur'pree) [from the LISP 1.5 (and later, MacLISP) function to start a new line of output] v. To output a CRLF (q.v.).

THEORY n. Used in the general sense of idea, plan, story, or set of rules. "What's the theory on fixing this TECO loss?" "What's the theory on dinner tonight?" ("Chinatown, I guess.") "What's the current theory on letting losers on during the day?" "The theory behind this change is to fix the following well-known screw..."

THRASH v. To move wildly or violently, without accomplishing anything useful. Swapping systems which are overloaded waste most of their time moving pages into and out of core (rather than performing useful computation), and are therefore said to thrash.

TICK n. 1. Interval of time; basic clock time on the computer. Typically 1/60 second. See JIFFY. 2. In simulations, the discrete unit of time that passes "between" iterations of the simulation mechanism. In AI applications, this amount of time is often left unspecified, since the only constraint of interest is that caused things happen after their causes. This sort of AI simulation is often pejoratively referred to as "tick-tick-tick" simulation, especially when the issue of simultaneity of events with long, independent chains of causes is handwaved.

TIME T n. 1. An unspecified but usually well-understood time, often used in conjunction with a later time T+1. "We'll meet on campus at time T or at Louie's at time T+1." 2. SINCE (OR AT) TIME T EQUALS MINUS INFINITY: A long time ago; for as long as anyone can remember; at the time that some particular frob was first designed.

TOOL v.i. To work; to study. See HACK (def #9).

TRAP 1. n. A program interrupt, usually used specifically to refer to an interrupt caused by some illegal action taking place in the user program. In most cases the system monitor performs some action related to the nature of the illegality, then returns control to the program. See UUO. 2. v. To cause a trap. "These instructions trap to the monitor." Also used transitively to indicate the cause of the trap. "The monitor traps all input/output instructions."

TTY (titty) n. Terminal of the teletype variety, characterized by a noisy mechanical printer, a very limited character set, and poor print quality. Usage: antiquated (like the TTY's themselves). Sometimes used to refer to any terminal at all; sometimes used to refer to the particular terminal controlling a job.

TWEAK v. To change slightly, usually in reference to a value. Also used synonymously with TWIDDLE. See FROBNICATE and FUDGE FACTOR.

TWENEX n. The TOPS-20 operating system by DEC. So named because TOPS-10 was a typically crufty DEC operating system for the PDP-10. BBN developed their own system, called TENEX (TEN EXecutive), and in creating TOPS-20 for the DEC-20 DEC copied TENEX and adapted it for the 20. Usage: DEC people cringe when they hear TOPS-20 referred to as "Twenex", but the term seems to be catching on nevertheless. Release 3 of TOPS-20 is sufficiently different from release 1 that some (not all) hackers have stopped calling it TWENEX, though the written abbreviation "20x" is still used.

TWIDDLE n. 1. tilde (ASCII 176, "~"). Also called "squiggle", "sqiggle" (sic--pronounced "skig'gul"), and "twaddle", but twiddle is by far the most common term. 2. A small and insignificant change to a program. Usually fixes one bug and generates several new ones. 3. v. To change something in a small way. Bits, for example, are often twiddled. Twiddling a switch or knob implies much less sense of purpose than toggling or tweaking it; see FROBNICATE.

UP adj. 1. Working, in order. "The down escalator is up." 2. BRING UP: v. To create a working version and start it. "They brought up a down system."

USER n. A programmer who will believe anything you tell him. One who asks questions. Identified at MIT with "loser" by the spelling

"luser". See REAL USER.
[Note by GLS: I don't agree with RF's definition at all. Basically, there are two classes of people who work with a program: there are implementors (hackers) and users (losers). The users are looked down on by hackers to a mild degree because they don't understand the full ramifications of the system in all its glory. (A few users who do are known as real winners.) It is true that users ask questions (of necessity). Very often they are annoying or downright stupid.]

UUO (you-you-oh) [short for "Un-Used Operation"] n. A DEC-10 system monitor call. The term "Un-Used Operation" comes from the fact that, on DEC-10 systems, monitor calls are implemented as invalid or illegal machine instructions, which cause traps to the monitor (see TRAP). The SAIL manual describing the available UUO's has a cover picture showing an unidentified underwater object. See YOYO. [Note: DEC sales people have since decided that "Un-Used Operation" sounds bad, so UUO now stands for "Unimplemented User Operation".] Tenex and Twenex systems use the JSYS machine instruction (q.v.), which is halfway between a legal machine instruction and a UUO, since KA-10 Tenices implement it as a hardware instruction which can be used as an ordinary subroutine call (sort of a "pure JSR").

VANILLA adj. Ordinary flavor, standard. See FLAVOR. When used of food, very often does not mean that the food is flavored with vanilla extract! For example, "vanilla-flavored wonton soup" (or simply "vanilla wonton soup") means ordinary wonton soup, as opposed to hot and sour wonton soup.

VAXEN [from "oxen", perhaps influenced by "vixen"] n. pl. The plural of VAX (a DEC machine).

VIRGIN adj. Unused, in reference to an instantiation of a program. "Let's bring up a virgin system and see if it crashes again." Also, by extension, unused buffers and the like within a program.

VIRTUAL adj. 1. Common alternative to LOGICAL (q.v.), but never used with compass directions. 2. Performing the functions of. Virtual memory acts like real memory but isn't.

VISIONARY n. One who hacks vision (in an AI context, such as the processing of visual images).

WALDO [probably taken from the story "Waldo", by Heinlein, which is where the term was first used to mean a mechanical adjunct to a human limb] Used at Harvard, particularly by Tom Cheatham and students, instead of FOOBAR as a meta-syntactic variable and general nonsense word. See FOO, BAR, FOOBAR, QUUX.

WALL [shortened form of HELLO WALL, apparently from the phrase "up against a blank wall"] (WPI) interj. 1. An indication of confusion, usually spoken with a quizzical tone. "Wall??" 2. A request for further explication.

WALLPAPER n. A file containing a listing (e.g., assembly listing) or transcript, esp. a file containing a transcript of all or part of a login session. (The idea was that the LPT paper for such listings was essentially good only for wallpaper, as evidenced at SAIL where it was used as such to cover windows.) Usage: not often used now, esp. since other systems have developed other terms for it (e.g., PHOTO on TWENEX). The term possibly originated on ITS, where the commands to begin and end transcript files are still :WALBEG and :WALEND, with default file DSK:WALL PAPER.

WATERBOTTLE SOCCER n. A deadly sport practiced mainly by Sussman's graduate students. It, along with chair bowling, is the most evident manifestation of the "locker room atmosphere" said to reign in that sphere. (Sussman doesn't approve.)

WEDGED [from "head wedged up ass"] adj. To be in a locked state, incapable of proceeding without help. (See GRONK.) Often refers to humans suffering misconceptions. "The swapper is wedged." This term is sometimes used as a synonym for DEADLOCKED (q.v.).

WHAT n. The question mark character ("?"). See QUES. Usage: rare, used particularly in conjunction with WOW.

WHEEL n. 1. A privilege bit that canonically allows the possessor to perform any operation on a timesharing system, such as read or write any file on the system regardless of protections, change or or look at any address in the running monitor, crash or reload the system, and kill/create jobs and user accounts. The term was invented on the TENEX operating system, and carried over to TOPS-20, Xerox-IFS and others. 2. A person who posses a wheel bit. "We need to find a wheel to unwedge the hung tape drives."

WHEEL WARS [from LOTS at Stanford University] A period during which student wheels hack each other by attempting to log each other out of the system, delete each other's files, or otherwise wreak havoc, usually at the expense of the lesser users.

WIN [from MIT jargon] 1. v. To succeed. A program wins if no unexpected conditions arise. 2. BIG WIN: n. Serendipity. Emphatic forms: MOBY WIN, SUPER WIN, HYPER-WIN (often used interjectively as a reply). For some reason SUITABLE WIN is also common at MIT, usually in reference to a satisfactory solution to a problem. See LOSE.

WINNAGE n. The situation when a lossage is corrected, or when something is winning. Quite rare. Usage: also quite rare.

WINNER 1. n. An unexpectedly good situation, program, programmer or person. 2. REAL WINNER: Often sarcastic, but also used as high praise.

WINNITUDE n. The quality of winning (as opposed to WINNAGE, which is the result of winning). "That's really great! Boy, what winnitude!"

WIZARD n. 1. A person who knows how a complex piece of software or hardware works; someone who can find and fix his bugs in an emergency. Rarely used at MIT, where HACKER is the preferred term. 2. A person who is permitted to do things forbidden to ordinary people, e.g., a "net wizard" on a TENEX may run programs which speak low-level host-imp protocol; an ADVENT wizard at SAIL may play Adventure during the day.

WORMHOLE n. A location in a monitor which contains the address of a routine, with the specific intent of making it easy to substitute a different routine. The following quote comes from "Polymorphic Systems", vol. 2, p. 54:

"Any type of I/O device can be substituted for the standard device by loading a simple driver routine for that device and installing its address in one of the monitor's 'wormholes.'*
----
*The term 'wormhole' has been used to describe a hypothetical astronomical situation where a black hole connects to the 'other side' of the universe. When this happens, information can pass through the wormhole, in only one direction, much as 'assumptions' pass down the monitor's wormholes."

WOW See EXCL.

XGP 1. n. Xerox Graphics Printer. 2. v. To print something on the XGP. "You shouldn't XGP such a large file."

XYZZY [from the Adventure game] adj. See PLUGH.

YOYO n. DEC service engineers' slang for UUO (q.v.). Usage: rare at Stanford and MIT, has been found at random DEC installations.

YOYO MODE n. State in which the system is said to be when it rapidly alternates several times between being up and being down.

YU-SHIANG WHOLE FISH n. The character gamma (extended SAIL ASCII 11), which with a loop in its tail looks like a fish. Usage: used primarily by people on the MIT LISP Machine. Tends to elicit incredulity from people who hear about it second-hand.

ZERO v. 1. To set to zero. Usually said of small pieces of data, such as bits or words. 2. To erase; to discard all data from. Said of disks and directories, where "zeroing" need not involve actually writing zeroes throughout the area being zeroed.

-*- Mode: Text; Fonts: CPTFONT,CPTFONTB,CPTFONTI; Hardcopy-Fonts: timesroman12,timesroman12b,timesroman12i -*-

Commonly Asked Questions:

***Q:* What are Logical Pathnames?**

A: [From the document "FILE," pg. 49] There is a kind of pathname that doesn't correspond to any particular file server. It is called a "logical" pathname, and its host is called a "logical" host. Every logical pathname can be translated into a corresponding "physical" pathname; there is a mapping from logical hosts into physical hosts used to effect this translation.

The reason for having logical pathnames is to make it easy to keep bodies of software on more than one file system. An important example is the body of software that constitutes the Lisp Machine system. Every site has a copy of all of the sources of the programs that are loaded into the initial Lisp environment. Some sites may store the sources on an ITS file system, while others might store them on a TOPS-20. However, there is software that wants to deal with the pathnames of these files in such a way that the software will work correctly no matter which site it is run at. The way this is accomplished is that there is a logical host called SYS, and all pathnames for system software files are actually logical pathnames with host SYS. At each site, SYS is defined as a logical host, but translation will work differently at one site than at another. At a site where the sources are stored on a certain TOPS-20, for example, pathnames of the SYS host will be translated into pathnames for that TOPS-20.

Here is how translation is done. For each logical host, there is a mapping that takes the name of a directory on the logical host, and produces a device and a directory for the corresponding physical host. To translate a logical pathname, the system finds the mapping for that pathname's host and looks up that pathname's directory in the mapping. If the directory is found, a new pathname is created whose host is the physical host, and whose device and directory come from the mapping. The other components of the new pathname are left the same. There is also, for each logical host, a "default device". If the directory is not found in the mapping, then the new pathname will have the same directory name as the old one, and its device will be the "default device" for the logical host.

This means that when you invent a new logical device for a certain set of files, you also make up a set of logical directory names, one for each of the directories that the set of files is stored in. Now when you create the mappings at particular sites, you can choose any physical host for the files to reside on, and for each of your logical directory names, you can specify the actual directory name to use on the physical host. This gives you flexibility in setting up your directory names; if you used a logical directory name called fred and you want to move your set of files to a new file server that already has a directory called fred, being used by someone else, you can translate fred to some other name and so avoid getting in the way of the existing directory. Furthermore, you can set up your directories on each host to conform to the local naming conventions of that host.

***Q:* Why can't I write out files (out of room) when I see that I have 30000 free blocks on my 474MByte disk?**

A: [From FILE, pg. 71] The 3600 disk is physically divided into partitions known as FEP files. This division of the disk is called the FEP file system. However, when one speaks of the file system of a lisp machine, one is generally referring to the LMFS (Lisp Machine File System) of that machine. This is the file system you edit when you click left on "Tree Edit Root" in the FSEdit window, and is the file system used when you specify file names of the form "*Lisp Machine Name:>directory>filename.type.version*". The entire Lisp Machine local file system normally resides inside one big file of the FEP file system (typically FEP0:>LMFS.FILE.1). Thus, LMFS is full when the amount of space allocated to it (in other words, FEP0:>LMFS.FILE.1) is full. Thus, LMFS could be full but there could still be 100,000 unused blocks on the disk (not even allocated as FEP files).

FEP files (for example, fep0:>Boot.boot) can be accessed from Lisp. The following files are part of the FEP file system. They should never be disturbed.

>disk-label.fep
>root-directory.dir
>free-pages.fep
>bad-blocks.fep
>sequence-number.fep

Other interesting FEP files include Microcode Loads, World Loads, Configuration Files, and Virtual Memory Files.

By convention, files of type .MIC are microcode loads. These files contain images of the microcode and the contents of other internal high-speed memories that are initialized when the 3600 is booted.

By convention, files of type .LOAD contain world loads (images of entire Lisp worlds).

By convention, files of type .BOOT are configuration files. Configuration files contain FEP commands tailored for a particular 3600 configuration. The commands are executed if you specify the file as argument to a Boot command when cold booting the machine.

By convention, files of type .PAGE are files used as Lisp Machine virtual memory while Lisp is running. Typically, there is one such file, FEP0:>Page.Page

Q: **I know to use "Tree Edit Root" in FSEdit to examine my LMFS. How do I examine my FEP filesystem?**

A: Simply use "Tree Edit Any", and specify a directory pathname of the form, "FEP*n*:>*", where *n* represents the disk drive containing the fep filesystem you want to edit. Also, the lisp function (PRINT-DISK-LABEL *n*) will print out a useful description of the FEP filesystem of drive *n*. (If you don't specify any *n* in the above, the "default drive" — the drive containing the world you booted from (see below) — is used.)

Q: **How do I make LMFS larger? Why isn't the rest of free space on the disk used for user files?**

A: [See FILE, pg. 68] Actually, a Lisp Machine's LMFS is not necessarily contained in just one FEP file. The file FEP0:>FSPT.FSPT tells LMFS which FEP files to use for file space, if not FEP0:>LMFS.FILE. To allocate more space, simply enter the File System Editor windo
w
(FSEdit), by using Select-F or by clicking on "File System" in the System Menu. Then click on "Maintenance". Answer "yes" to the warning question about taking responsibility for your actions. Then click right on "Initialize". A little menu will pop up. Click on "Auxiliary Partition" and click on the name above this so that you can specify a name for the auxiliary partition. Typically, a good name is FEP0:>LMFS-AUX.FILE. (Of course, if you have more than one drive, or a FEP file named LMFS-AUX.FILE already exists, this may not be a good name.) Then click on "Do It". It will ask you how much space to allocate to this file; specify a number of blocks. It is intelligent to specify this number in decimal, with a trailing decimal point, as in **30000.** if you want thirty thousand more blocks in your LMFS.

Q: **How do I create a FEP file?**

A: Why? There aren't too many reasons for creating FEP files, but, if you have a good one, the right way is NOT to use the FSEditor window. Many users have assumed that since the "Initialize" option described above creates a named FEP file of a specified size for you, that this is an acceptable way of creating FEP files. In fact, this is quite the opposite. The FSEditor is only good for creating FEP files that should be allocated to LMFS. If you create a file in such a fashion and then use it for something else, the LMFS data structure contained on your disk may become very confused, and can potentially destroy the file system of your machine. The following "incantation" will suffice to create a FEP file for you.

```
(WITH-OPEN-FILE (FILE "FEPn:>Filename.type.version"
                 :DIRECTION :BLOCK
                 :IF-EXISTS :ERROR)
   (SEND FILE :GROW 30000.))
```

(Where the italicized string above represents the name of the FEP file to be created, and the italicized 30000. represents the size you want to make the file. I used 30000. with a decimal point to remind you that it is intelligent to use the decimal point and think in base 10 for this kind of operation.)

***Q:* What is a world load?**

A: A world load can be thought of as a "snapshot" of an operating lisp environment. All of the functions, variables, and other lisp objects that were present in the lisp environment when the snapshot was made are contained in the world load file on the disk. Typically, snapshots of worlds are made only when such a snapshot would save significant time later. For example, after you've initially configured your new machine at your site, it is useful to make a snapshot of the configured environment because it will save you time in the future (you won't have to configure the machine each time you boot it). If you usually load Macsyma or Fortran each time you boot, it may be advantageous to make a snapshot of a world with that software loaded, to save you the time of loading it. Remember, everything in the environment is contained in the snapshot, so you don't want to create a world load file after you've been using the editor or most system facilities (you don't want to find old text in your editor buffer when you boot, do you?). The way to create a snapshot and save it to disk is by using the function (DISK-SAVE) [which can also be invoked as (DISK-SAVE *"FEP:>Filename.LOAD"*)]

***Q:* What happens when my machine boots?**

A: The command "BOOT" or "B" to the FEP means "Read in a configuration file and execute t
he
lines in it sequentially as FEP commands." The default is displayed for you if you type a space after the word "boot" or "b". The default configuration file when the 3600 is powered up is FEP0:>boot.boot. After that, the default configuration file is the last configuration file booted. You can give an argument to the BOOT command of a boot file to boot from. You must specify the name fully (as in ">Rel5.boot", "Rel5" is not enough, as the FEP cannot merge pathnames).

A configuration file usually contains FEP commands to:

- Clear the internal state of the 3600
- Load the microcode
- Load a world
- Set the Chaosnet address
- Start the 3600

To change the selection of microcode and world loads that are booted by default, simply use Zmacs to edit the file >Boot.boot. Be careful to avoid typographical errors; otherwise, you might have to type in the commands manually in order to boot the machine. Also, be sure that the last command in the file is followed by RETURN.

A more detailed explanation follows:

The FEP command Load World copies a portion of a world load into the paging area. The Start command sets the Lisp program counter at a fixed address (where the function SYSTEM-STARTUP resides) and says "Go". Lisp takes over from there.

***Q:* Why do we name hosts, printers, etc?**

A: Naming inanimate objects such as hosts, printers, sites, and networks may seem foolish to a customer with zero or one of each, but for customers with large numbers of machines, names are a convenient way to easily refer to a particular machine with a particular address without you having to remember the address, machine type, etc. Besides, it can be a lot of fun to argue with your co-workers about the names for your machines. Here at Symbolics, we name all of our 3600's and 3670's after rivers, and all of our 3640's after birds. One customer named its machines after the characters in Winnie the Pooh, while another named its after the wives of Henry VIII.

***Q:* What is a "system?"**

A: [See MAINT.] A system is simply a named collection of files. Symbolics software products such as FORTRAN, Interlisp, and Color are all systems. Many of the core elements of the operating system are systems, as is the operating system itself. Groups of files are defined as a system for a number of reasons. The primary reason is for control. It is easy to ensure that files get compiled in the correct order if they have been defined as a system. Also, distributing the software to other locations (as we do with our software packages) is easy if the software has been defined as a system. If you have one of our packages, you'll see how easy it is. Once you've loaded the tape (using (DIS:LOAD-DISTRIBUTION-TAPE)), all you need to do is (MAKE-SYSTEM '*system-name*) to load the
software.

```
;;; -*- Mode: TEXT; Package: USER; Syntax: Zetalisp; Lowercase: T; Base: 10; Fonts: CPTFONT,CPTFONTCB,CP
TFONTI; Hardcopy-Fonts: FIX9,FIX9BB,FIX9I -*-
;;; Created 1/23/85 18:40:39 by sgr
```

Guidelines for Demonstration Software Developers -or- Rowley's Rules of Order

by

Steve Rowley

These guidelines are meant for those of you who want to develop demonstration programs, or re-tool old programs so that they're in shape for being demonstrated. We invented these guidelines on the basis of several years experience in integrating software for technical shows, world-building, and so forth. The bottom line is that if your software follows a few simple rules, we'll be able to bring it up and integrate it into a world with other demonstrations with a lot less frustration for both of us.

If you have trouble understanding any of these rules, feel free to call your software service person. If, *for very pressing reason*, you feel we should make an exception in your case, feel free to call the show coordinator. He may feel that he can make a decision immediately, or he may want you to talk to the technical person in charge of software integration. Exceptions will be rare, such as software we are already very familiar with from previous experience. In any case, be sure to *allow plenty of time*. Last-minute requests for exemptions will be looked on askance, because we're probably going just as crazy as you are.

Rule 1: Your software must be a *"system"*, as described in the MAINT section of the documentation. That is, we must have only to say *(make-system <your-system-name> :noconfirm)* to bring it up. We should not have to type in any magic forms, expose any windows, load any extraneous files, or anything else of that sort. This means you will have to provide a *<your-system-name>.system* file, a *<your-system-name>.translations* file (discussed in **Rule 2**, below), and a system-declaration file.

Rule 2: Your software must use *logical pathnames*, as described in the FILE section of the documentation. One of the most frustrating problems someone integrating your software into a world load faces occurs if your software suddenly reaches out for a file on some VAX at your site, but can't find it because it's not *at* your site anymore. Logical pathnames provide a way of referring to files in a site-independent way. The *<your-system-name>.translations* file provides us with a way of informing your program where its files are now kept.

Rule 3: Your system should create and stay inside its own *package* as much as possible. This is so that if your software and someone else's define a function called *foo*, there won't be any conflict. Packages are described in the PKG section of the documentation. (If, for some arcane reason, you feel your software *must* affect symbols in standard packages (e.g., USER), then you must document the effect for us. However, we may not load your software if we think you're doing something potentially unfriendly to other programs.)

Rule 4: Your software should not redefine system symbols. If you don't like the behavior of, say, *loop*, then you should *shadow* it in your own package (see **Rule 3**, above) rather than redefine or patch it globally. This is because your patch may cause other people's software to get violently ill; other people have a right to continue to expect system functions to operate as advertized, whether or not your software is loaded. Shadowing is discussed in the PKG section of the documentation.

Rule 5: There are a number of global switches that affect the behavior of Lisp, such as base, ibase, *nopoint, and so on. Feel free to *bind* these as much as you like, but please do not *set* them (at least not permanently, or outside an *unwind-protect*). You may like to use hexadecimal input in your program, but other software may not like that.

Rule 6: There is no **Rule 6**. This is not described anywhere in the documentation, so we thought we'd say it here.

Rule 7: Your software must be written in the "announced release" of the show. For example, if we announce that a certain show will be done in Release 5.2, then all software must be in 5.2. We will not accept or offer to convert demonstrations in Release 4.5, or 5.1, or 6.0, or whatever. If you don't have the announced release at your site, contact your local service office to have it installed.

Rule 8: You must provide *source code*. We will sign non-disclosure agreements if necessary, and will take all reasonable measures to see that nobody steals your stuff. However, if you don't provide source, we can't recompile to fix any interactions with other software that we discover. For example, at one show, a customer compiled all his software on our system and then did a wildcard delete of *.lisp in his directories. This inadvertently killed the system declaration file, so we couldn't make his system after he left. His software was not shown.

Rule 9: You should expect to run in a world where the garbage-collector is turned on. In Releases 6.0 and later, the ephemeral garbage-collector will be on also. You may not turn it off for your demo. This is to avoid re-booting machines as much as possible.

Rule 10: We should be able to *compile* and *load* your system in the announced release (see **Rule 7**, above) with *no warnings whatsoever*. This means you may have to stick in some shadowing, special declarations, and the like. However, it means we can rely on your program's likelihood of being correct much more than otherwise.

Rule 11: Your software must not leave any *"time-bombs"* in the machine either after loading or being run. For example, some programs that use special subprimitive operations (see the INT section of the documentation) mistakenly violate system storage conventions. When the garbage collector (see **Rule 9**, above) finally sees such nonsense, it usually ends up halting the machine. This is considered impolite (at least).

Rule 12: Your software must be *able to be loaded into a world where it's already present.* (We may need to re-load your software after someone else's software clobbers part of it. We don't want to have to re-build an entire world to do that.) Some systems can't do this because, for example, they create a window of which there must always be at most 1 instance. Others might construct circular lists at load time which throw a subsequent load into an infinite loop. *Be prepared.* Don't assume your window isn't there, the list isn't circular, or whatever.

Rule 13: Last, a rule *so important* that we gave it the lucky number 13. *The only form in which we will accept delivery of your software is a distribution tape written at standard densities on cartridge tapes.* In particular, world-load tapes, carry tapes, LMFS-dump tapes, and others *will be returned unread*. Distribution tapes are discussed in the TAPE section of the documentation. Make sure your tape and its plastic box *both* have your software's name, and the name, address, and phone number of a "contact person" in your organization. Also mark the cartridge to tell us if you want it returned to you after the show or not. Unlabelled tapes tend *to get lost* in the hectic environment of a show. For each show, we will announce a *"cut-off date"*, by which all tapes must be *in our hands*. If you send us a tape after the cut-off date, we may or may not load it, depending on how much time we have. You should not count on *any* time at the show site for bringing up your software; this must all be done in the staging period in advance of the show. At the show we will deal only with very serious problems. Late demos are not in that class.

-*- Fonts: CPTFONT,CPTFONTI,CPTFONTCB; Hardcopy-fonts: timesroman12, timesroman12i, timesroman12b -*-

Technical Application Note

Tuning programs for the 3600.

By Eric Weaver, Symbolics Graphics Division

-1. Introduction

This paper is presented as an attempt to distribute some knowledge of the art of "code bumming" for the 3600. Some of these techniques have been an "oral tradition" for years, some have been in the documentation in obscure places, and so on. I hope to get the maximum number of these all together for our users.

0. Architecture Hacks

The 3600 has an architecture which was designed to run Lisp, but certain kinds of programming practices are more advantageous than others. Here are some hints for best use of the machine..

A. Using locals instead of free references and constants

Local variables ("bound references" inside compiled funcions) are at least 5 times as efficient as *special* variables ("free references"). Locals are stored in the hardware stack frame, and the instructions to fetch, store and manipulate locals are very efficient (one cycle in the best case). Special variables, on the other hand, require two memory references, each of which takes four cycles. Binding a local to the value of a special variable in an inner loop will save much.

Also, binding a local to a constant is about 4 times as efficient as referring to the constant directly (this is due to the way constants are kept in the compiled function). This is true for all constants *except* for **nil** and small integers (-128 to 127).

Locals can be made with **let**, **destructuring-bind**, **prog**, and any number of other mechanisms (including **&aux** variables declared in the lambda-lists of functions).

By the way, it is faster to do a **let** outside a loop and use **setq** within than to have the **let** inside the loop (due to overhead of pushing and popping things inside the loop). Also, incrementing, decrementing and **cdr** of a local variable are especially fast.

B. Array Registers

An array reference usually involves several memory accesses. This is to find the type of the array, its dimensionality, its upper bound, and data address. This is called *decoding* the array. One can, by using *array registers*, decode the array once, and save all that information in the hardware stack for repeated use (see the section above on local variables; the same principle is at work here).

An array register is essentially a local variable, which is treated specially by the compiler. It takes up four words in the stack, and stores the information about the array. Array references using the array register are done using a faster kind of array-reference instruction, which uses the information on the stack. Although the decoding of the array is done once, full bounds checking is still done.

The amount saved by array registers depends on the type of array. It's not much for small, simple arrays with no leader. On an array with a leader, or on an indirect or displaced array it saves a lot. Certain non-word-aligned displaced cases are actually made *slower* by array registers, currently. Examples of indirect arrays in the system are network packets and window-system screen arrays.

An array register *must* be bound with **let**, and must be declared to be an array register using the **declare** form. The array register must *never* be set (using **setq**, et al.), only referred to. They work for one-dimensional arrays, and for "linearized" cases of multi-dimensional arrays (such as one would use **%1d-aref**). The same restrictions apply to uses of array registers for multi-dimensional arrays as to **%1d-aref**. To use this feature with arrays that may be conformally indirected, use function **si:array-column-span** to determine the distance in elements between columns (or rows, if you're using graphics notation).

Example:

```
  (let ((array *global-array*))
    (declare (sys:array-register array))
    (loop for i from 0 below (array-length array)
          as summing (aref array i)))
```

Note: It does *not* work to use array registers with a **loop** iteration path. In the following example:

```
  (loop for x being the array-elements of array ..)
```

loop binds its own local variable (an uninterned symbol, like **#:G0234**) to the array, and does all references using that variable.

sys:array-register	*variables...*	*Declaration*
sys:array-register-1d	*variables...*	*Declaration*

These declare variables to be array registers. When the binding is entered, the array is decoded and the information saved in the array-register associated with the local variables. **array-register** will require a one-dimensional array, while **array-register-1d** will accept any dimension of array and force it to be decoded "linearized". One-dimensional references to that array may be compiled with the **fast-aref** instruction, and will execute much more quickly than ordinary array references. If a **setq** to the variable occurs in the binding, the compiler will signal an error (that applies to Release 5.2; in older systems, incorrect code will be compiled!). It *does not* work inside a **loop** macro to so declare a variable bound by the **loop**.

This feature is not currently in the documentation, but is in the Advanced Lisp Course notes.

C. Branching

On machines without the Instruction Fetch Unit (which at this writing is *all* machines), branch instructions which *do* branch take at least 50% longer than *don't* branch. One can gain speed by arranging programs to branch as little as possible in the fast case.

Sometimes this requires using (ugh) **prog**, sometimes it merely requires the reversal of clauses or of the sense of a condition.

One can use the **disassemble** function or the editor command Disassemble (m-X) to see the kind of code being generated, and use that to determine whether the fast case is taking the branch or not.

D. Open Coding and substs

A function call takes the approximate equivalent of ten instructions. Open coding small functions in loops could save a lot.

A **subst** is a function which can be open-coded by the compiler when found in a function being compiled. This gets part of the benefits of both macros and functions. For more on this, see the section on Substitutable Functions in Chapter 3 in the Macros document in Volume 3.

E. Other random things.

Some primitive functions (those which are implemented by machine instructions, for example +, -, etc.) are more effective when the last argument is a local variable. For example, (+ (foo x) y) generates one fewer instruction than (+ y (foo x)).

ldb and **dpb** are much faster (one cycle) when the byte field is constant (and suitable for a fixnum result). (ldb ■o2020 foo) is much faster than (logand foo ■177777), due to the time needed to push the constant.

1. Not Consing

Allocating memory takes time, and (of course) eats up memory, making it necessary to garbage collect sooner. Some hints for not consing...

A. **push, collect, nconc** instead of **append**

Using **append** in loops can be quite wasteful. Particularly if you're using it to append ONE thing to a list. Here are a few ways do collect lists of things without excess consing:

push is a macro which conses a new first element on to a list. It expands thusly:

```
 (push item list) → (setq list (cons item list))
```

This collects the list "backward", i.e. the last item collected will be first on the list. If that's not acceptable, there's more..

collect (also **collecting**) is a clause for the **loop** macro. For greater detail, see the Macros document in Volume 3. Use of **collect** will cons only once for each new element collected, and use a minimum of cycles to do it. The list is collected "forward", i.e. the last item collected will be last on the list.

Example:

```
  (loop for i from 0 to 99.
        collecting i)  =>    (0 1 2 3 4 ....)
```

nconc is a function which is the destructive version of **append**. It actually changes the last CDR of each list appended to, to form a single list containing all the elements of all the lists. This should not, however, be used lightly. *Don't* use **nconc** if any of the lists appended to is one you will want to use again, as they will be altered.

nconc (also **nconcing**) is also a clause for the **loop** macro. For greater detail, see the Macros document in Volume 3.

Likewise, any of the list functions prefixed with "n" perform actions directly on the list structure, without copying. The same cautions apply.

B. Arrays instead of Lists

Some applications actually require a small amount of storage, which can be re-used, rather like a stack. An array may be used for such. For example:

```
(defconst *stack* (make-array 100. :fill-pointer 0))

... (array-push-extend *stack* thing) ...
    (let ((thing  (array-pop *stack*)))
      ... ) ...
```

This technique does all the consing once, and uses a very quick mechanism for allocating the cells. It is also effective at keeping the elements together in the address space (see Locality Control below).

This is documented in the Arrays document, in Volume 3 (except for the **:fill-pointer** keyword which is mentioned in RN somewhere).

C. Resources

If you are consing things which could be re-used, you may want to keep them around after you're done with them. See the document on Storage Management in Volume 8.

D. Things that cons behind your back

Some parts of the system will cons madly when you use them. Here are some:

On systems earlier than Rel. 6, SQRT conses bignums up a storm. Avoid using it in loops unless you have to.

On systems earlier than Rel. 6, DPB may make a bignum if you try to mess with the sign bit. Use %LOGDPB, which returns fixnums at all times.

Extended-number arithmetic conses all results and occasionally conses intermediate results, although it tries to avoid that when it can. This includes bignums, ratios, double-floats, and complexes.

Both SUBSTRING and NSUBSTRING cons. The break-even point is 4 characters (was 12 characters on the LM-2). This is why lots of string functions and the :STRING-OUT message take optional subrange arguments.

(Contributions here, please...)

E. Stack Lists

You can compose a list which is kept entirely on the stack, and is de-allocated when the function containing it returns. The list takes up no storage outside the stack. Caution: Do not try to store a pointer to the list anywhere permanent, as the pointer will point to garbage when the function returns and the list is deallocated. This may be fixed someday.

with-stack-list *variable &rest elements* *Special Form*

Makes a list of *elements* on the stack, and binds *variable* to it. The length of the list is, unfortunately, a declarative attribute (not specifiable at runtime).

with-stack-list* *variable &rest elements* *Special Form*

Like the former, only the last cons is "dotted" (the last value passed goes in the last CDR, not the last CAR), just like using **list***. Useful when temporarily "pushing" things on the front of a list (the error system uses it in that way when binding condition handlers).

2. Locality Control

Paging performance can be improved by keeping objects together which are used together. A list which is threaded throughout the address space will take much more paging to work with than one kept all together.

A. Compact Lists

Traditionally, a list is represented as a string of two-word conses, storing *n* elements in *2n* words. However, it is possible to make *compact* lists which take up only *n* words, and are consecutive in memory. Thus, after a list is composed, using **cons** numerous times, one can copy it and store the compact copy in place of the old non-compact one. This is done by the flavor system, for example, in keeping "which-operations" lists for flavors. **list** and **copylist** make compact lists. For greater detail, see the section on "CDR-coding", in the Primitive Object Types document in Volume 3.

B. Areas

An *area* is a logical piece of the virtual address space in which one can allocate storage. The address space is normally divided up between 30 or so areas, each for different uses. The reason for areas is to keep logically similar (or similarly used) objects together in the virtual address space. Flavors are made in one area, Editor lines are kept in another, etc. This way, they are all close together in the address space although they may be allocated at different times.

Normally all consing takes place in an area called **working storage area**. A program which conses at various times will have objects scattered throughout that area. This could result in needing many pages to be paged in to access the few object your program needs (this number of pages is called your program's *working set*). Consing some of your objects in a specially created area can improve paging performance markedly in some cases.

An example application of areas is Sage, the on-line documentation program. It keeps many strings from various files, all of which it must search upon request. It keeps the strings in a special area, which is kept together in virtual memory. When it searches through the strings, much less paging is required because fewer pages are referenced in the process.

The ideal candidates for area hacking are:

1. A program which sometimes conses up things which are essentially permanent. Sage's strings are like this.

2. A program which conses up things which become garbage fairly often, and one would like to GC them separately from the rest of the system. This may be unnecessary now with EGC, but it was true in the "old days".

Anything which can be consed can be consed in a specific area. Functions **cons-in-area** and **list-in-area** allow the user to specify where to make lists, **make-array** takes a keyword argument **:area**, and the special variable **default-cons-area** can be bound to a specific area, to direct the consing of everything whose area is not specified.

For more detail, see the section on Areas, in the Storage Management document in Volume 8.

3. Rolling Your Own

Not all system functions and utilities do the most efficient thing for your application. Sometimes you will have to take either a more specific or a more general approach. Some notable areas to which this applies are listed here.

A. Hash Functions

Using EQUAL hash tables is not always the right thing when hashing large trees (such as forms and rules), as the hashing function (**si:equal-hash**) will try to traverse the entire tree and uniquely characterize it. This can lose the some of the efficiency of hashing.

When you know what parts of the tree (or list) will characterize it, you can make a flavor of hash table which hashes on only those parts. If you knew you could hash very well on just the first three elements and the length, you could then make a hash table to hash that way.

Beware, however, of the possiblity of unequal items hashing to the same value (which can ruin the value of the hash table). The hashing employed must be based on the characteristics of the kind of objects being hashed.

Examples for ways to code hash functions may be taken from the code for **si:equal-hash**.

A (1). Making your own flavor of hash tables

You can make a flavor of hash table which uses your hashing function. Here is some reference material:

si:generic-hash-table *Flavor*

This is a flavor of **equal-hash-table** which requires a specific hash function and comparison function to be supplied, as methods for this flavor. These are specified below. This is an abstract flavor, and should be mixed in with your own protocol for this kind of hashing.

Your own protocol must provide the following two messages:

:hash-item *item* *Message to* **si:generic-hash-table**

Given the item, hashes it and returns two values:
1. The hash code for the item as a fixnum (preferrably a non-negative one; although negative ones work, they're much less efficient).
2. A boolean value (T or NIL), true if the hash function depends on the object's address (%pointer) and hence may need to be recomputed after a garbage collection (the second value may be omitted if it's always NIL).

:equal-items *item-1 item-2* *Message to* **si:generic-hash-table**

Given two items, must determine whether for the purposes of this hash table they are equal. You may use EQUAL if that applies, or do your own test.

The following code fragments show a mixin which implements that protocol, and a final flavor built from it and **generic-hash-table.**

Example:

We know a "rule" is a list that looks sort of like a lisp form. For example...

```
 (cousins ((*var* a) (*var* b)) <=  .  <list-of-long-hairy-forms>)
```

The first element is the name, the second is sort of a Lambda-list, and the third is the implication specifier. A reasonable characterizer for a rule is to hash the first three elements and add the length for good measure.

Here are some code fragments to do that...

```
;;; Build our own kind of hash table for hashing rules better.
(defflavor rule-hashing-mixin () ()
  (:documentation
   (:mixin "Proper hashing for rules: first three elements and length.")))

(defun hash-first-three-elements-and-length-of-list (list)
  ;; Given a list, hash the first three elements and length into a hash
  ;; key.  Faster than EQUAL-HASH on the list for rules and other hairy
  ;; lists.
  (let ((first (equal-hash (first list)))
        (second (equal-hash (second list)))
        (third (equal-hash (third list)))
        (length (equal-hash (length list))))
    ;; The following technique for hashing lists seems to be
    ;; in line with the philosophy of the code in EQUAL-HASH.
    (non-negative-fixnum
      (logxor (rot first 5.)
              (rot second 10.)
              (rot third 15.)
              (rot length 20.)))))

(Defmethod (rule-hashing-mixin :hash-item) (item)
  "Return the hash number for this item (presumably a rule)"
  (cond ((listp item)
         (hash-first-three-elements-and-length-of-list item))
        (t (equal-hash item))))

(defmethod (rule-hashing-mixin :equal-items) (x y)
  "Predicate whether the things are equal for our purposes.  Same as EQUAL."
  (equal x y))


(defflavor rule-hash-table () (rule-hashing-mixin
                                si:generic-hash-table)
  (:documentation
   (:combination "A hash table for rules.")))


(defconst *rule-hash-table*
          (make-instance 'rule-hash-table)
  "The hash table to hash the rules in")
```

This is currently not in the documentation. The code is in "SYS: SYS2; HASH.LISP".

4. Traditional techniques

Compilers for other languages often can do very hairy code optimiztion. The Lisp compiler, however, does not do any except numeric constant folding. Users should therefore be aware of these common efficiency measures...

A. Code hoisting (out of loops)

Moving constant computations out of loops is called (in compiler jargon) *code hoisting*. Anything which is done in a loop which is defined to yield the same result every time may be put in a local variable outside the loop, and the value referred to.

B. Common sub-expression merging

Separate expressions which contain common parts could share the computation of those parts. Binding a local to the intermediate value and using it in the expressions saves extra computations of that value. **let** and **let*** are useful for this.

5. Metering techniques to help find the hogs

The problems are not always obvious. Clearly, if you do much data manipulation and the "page bar" is on much of the time, you can fairly well guess that there are locality problems. Other kinds of inefficiencies may not be so easy to spot. Here are some techniques for metering your programs to find where they spend their time.

A. Breaking

Sometimes, simply pushing c-m-Suspend every now and then, and looking at the function that is running at the time, can tell much about the bottlenecks in a program. If the program has large areas run **without-interrupts**, however, this is not effective.

B. Stack Metering

This is a special system, which uses the microsecond clock to record the time between entry to and exit from functions. It can be run on any application, and show the time, consing and paging done by each function, both inclusive and exclusive of the functions it calls.

Where is this documented, anyway?

It doesn't seem to be distributed, so maybe this doesn't belong here.

C. PC Metering

This can give an analysis of which functions are taking up the time in a program. It's documented in Release Notes, in the section called "New Metering Tools for the 3600."

D. Increment Metering

DLW mentioned this but I must not have understood it.