DRAFT - Edit Me

SUN

User's Guide

11 December 1981

Editor: W.I. Nowicki

Contributions by: J. C. Mogul and V.R. Pratt

Table of Contents

List of Figures

1. Introduction

SUN is the *Stanford University Network*. This is an on-going effort of several groups within Stanford University to provide a local computer network for Stanford University [2], based on inexpensive but high performance workstations [3]. The network currently connects several time-sharing systems [7] in the Computer Science and Electrical Engineering Departments, with plans to extend the network to the rest of the campus [16]. The processor architecture in the workstation is the Motorola MC68000. The workstation uses the Intel Multibus[1], which is proposed IEEE standard 796, so many other common peripheral interfaces are comercially available. "Workstation" is used instead of "personal computer" or "terminal" to emphasize the flexibility of the design.

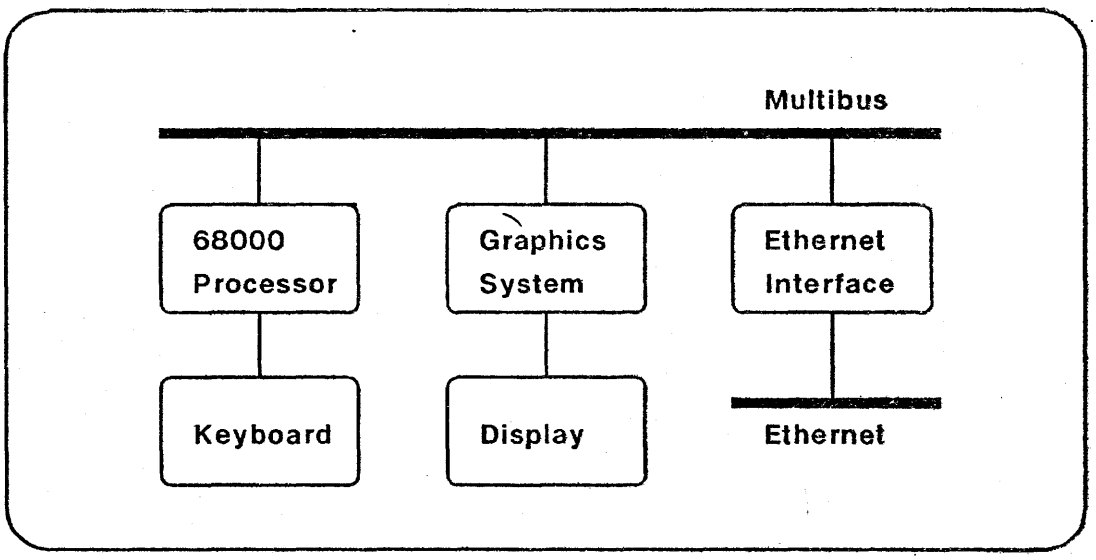

Figure 1-1: Major SUN Workstation Components

There are currently several operating systems being developed for the SUN workstation. *Perseus* [11] is a distributed operating system based on messages and links [4] that has been evolving for several years. It is written in Pascal* [8], a version of Pascal for systems programming. In the mean time, there are several attempts to bring up Unix in various stages. Vaughan Pratt has written his own stream-based operating system (see Chapter 7). Another operating system is being developed to be used in a multi-processor configuration.

[1] Multibus is a trademark of Intel Corporation

1.1. Intended Audience

This manual is intended for people who want to write programs for the SUN workstation. Eventually this should be called the "Hacker's Manual," with a separate manual sparing casual users of the unnecessary implementation details. The emphasis is on getting consistent up-to-date information, rather than a pretty manual. Suggestions, additions, editors, and proofreaders are welcome. Many of the tools described here are preliminary or interim in nature. Since everything changes so fast, you can incrementally generate any of the chapters of this manual.

The source for this document is stored in `[Shasta]/usr/sun/doc`. Each chapter exists as a press file, which can be printed with the `cz` command.

```
cd /usr/sun/doc
cz  <chapter>.press
```

There is a makefile in that directory, so a command of the form

```
cd /usr/sun/doc
make <chapter>.press
cz  <chapter>.press
```

will make a given chapter into a press file, which is then be printed on the Dover printer. The `cz` program automatically inserts the figures in their proper places. To get the entire manual, simply use the commands:

```
cd /usr/sun/doc
cz manual.press
```

1.2. Related Documentation

1. The architecture of the MC68000 procesor and a description of the processor chip can be found in Motorola's User's Manual [13]. There are several books describing the 68000 architecture [9] [10]. Most vendor-supplied hardware is described in separate manuals or data sheets.

2. The hardware manual [1], currently somewhere on Sail, describes the processor, ethernet, and graphics boards designed at Stanford. Someday we should convert this into Scribe and merge it into this manual.

3. The Documentation for Unix[2] can be found in the standard Unix [15] and Berkeley manuals. Most of this is stored on the large Unix systems, and available through the `man` and `apropos` commands. Some specific MC68000 Unix commands are described in the next Chapter.

4. The Perseus documentation, under `[IFS]<Perseus>`. Perseus is a distributed operating system, currently under development. There are probably a dozen CS246b projects that are related, the documentation of which is scattered throught the world.

---

[2] Unix is a trademark of Bell Laboratories

2. Software Development on Unix

Currently our development work is being done on a VAX[3] computer running the Unix operating system. With any luck, Unix may soon be running on the 68000 itself.

2.1. Organization of SUN directories

Common SUN software is stored in subdirectories under one master directory. For example, on the Stanford VAXes Shasta and Diablo, this directory is `/usr/sun`. The following is a description of these subdirectories, using the notation relative to `/usr/sun`, or whatever it happens to be called on your system. Every source file should contain a comment near the begining describing the author, date, and purpose of the file.

./admin	Administrative records, and a wishlist.
./bin	Binary files for Sunix commands. Complain to Vaughan about documenting **Sunix.**
./bootfile	Standard boot-format files. The production copies of stand-alone programs reside here. This is the default directory for the SUN boot server. "Test" versions are put in the subdirectory `test`, not in the main directory.
./dm	Files to support the Motorola Design Module. Includes `.dl` format files of some programs that run on the design module. Read chapter 5 for more information on the support for the design module.
./dm/lib	Object library files specific to the design module.
./dm/include	Header files specific to the design module.
./doc	Overview documentation. The document you are reading now resides here. Some other documentation exists in subdirectories of this.
./doc/graphics	Description of a Raster-op level graphics package.
./include	Header files, used by the `#include` directive of the C preprocessor. See section 2.3 for more information.
./lib	Object libraries. These are searched by the `-l` option of `cc68`. Some of these are described near the end of this chapter.
./man68	Master copies of manual pages describing Unix commands. See next section for details on these commands.

[3] VAX is a trademark of Digital Equipment Corporation

./monitor `.dl` files for making monitor proms. Sources are hidden under ./src/monitor.

./src Some sun-related sources, subdivided into the following subdirectories. These should correspond to the debugged, "production" versions of programs. Each subdirectory should have a `makefile` describing how to compile sources in it; the result of the `makefile` should be the bootfiles in ./bootfile or ./bin.

./src/cmd Commands that run on Vax and Sun. Further divided into subdirectories for some of the important tools like the compiler, loader, etc.

./src/diag Sun hardware diagnostics. These consist of a memory test, an ethernet interface test, and a graphics board test.

./src/games Source for some games. Most of these run on a "dumb" terminal, instead of using the graphics board.

./src/graphics Graphics demostration programs, and a Raster-op package.

./src/libc Sun libc.a sources. Further subdivided into the following subdirectories.

./src/libc/crt C run-time support. This is the routine which sets up the stack and calls the `main` function of a program.

./src/libc/emt Emulator traps into the PROM monitor.

./src/libc/gen General functions. This includes string manipulation, string conversion, etc.

./src/libc/stdio Standard IO functions. Currently this all goes through the terminal line.

./src/libc/sunstuff SUN Processor specific part of the library. Isn't all this stuff? Please complain to vaughan about this.

./src/libc/test Floating point tests. Has anybody ever used this?

./src/libc/unixstdio
Some Unix stdio stuff, never used for anything.

./src/monitor PROM Monitor sources. Complain to Jeff that the up to date versions are kept in his private directory instead of the proper place.

./src/mut Multi-user Pup Telnet. This is the program that runs on "Ether Tips", and in a multi-window terminal program located in ./src/tty.

./src/tty Multi-window terminal program.

./unix68 Nu terminal Unix from MIT - Optional. We should put the LucasFilms Unix there someday, if we ever get it.

2.2. Unix Commands Relevant to 68000 Software

The following Unix commands are used for MC68000 software development. All of them are documented in chapter 1 of the Unix manual. This listing may be obtained using the Unix command `apropos 68`.

as68(1)	- Assembler
cc68(1)	- General C command
ccom68(1)	- Portable C compiler
ddt68(1)	- A symbolic debugger and disassembler
dlx(1)	- Down line load protocol handler
dl68(1)	- Download file generator
ld68(1)	- Linking loader
lorder68(1)	- Object library utility
nm68(1)	- Print name list of object files
o68(1)	- optimizer for assembly language
pc68(1)	- Pascal* compiler (similar to cc68)
pr68(1)	- print extended statistics on .b file
rev68(1)	- reverse byte order .b and .68 (b.out) files
rl68(1)	- print relocation commands in a .b file
size68(1)	- prints sizes of segments in a .b file

2.2.1. Compiling with CC68

These commands are meant to mirror the standard Unix commands without the "68" suffix. For example, the normal C command is `cc`, and the coresponding MC68000 command is `cc68`. This is the command most users will be concerned with. Its function is to take the files named as arguments and do whatever needs to be done to make them into a runable program. For `cc68` to work properly, you should follow some simple naming conventions. File names should consist of a short module name, followed by a suffix consisting of a dot and one or two letters. The suffixes are listed below:

`.c`	C language source programs. These are typed in and edited by the user with any editor.
`.p`	Pascal*. (or regular Pascal) source files. See the Pascal* reference manual [8] for a description of the language.
`.h`	Header files, usually consisting of declarations and macro definitions which are accessed by the `#include` directive. Some useful header files are described in section 2.3.
`.s`	Assembly language files, produced by the compiler or written by hand. Compiler produced `.s` files are usually not seen by the user, except for detailed optimization or debugging. These used to be called `.a68` files, but `cc68` used `.s`, which unfortunately conflicts with all other assembler files, such as VAX assembler files.
`.b`	Binary files, the output of the assembler.
(none)	The default output of the linker is the file name b.out Usually an explicit output file will be specified with the `-o` option, the name of the program without any suffix. If the file is

generated on a machine with non-standard byte order such as a VAX, the `.r` format should be the final result.

`.r` Byte reversed files produced by `rev68`, ready to be loaded over ethernet. In actual practice, the `.r` suffix tends to not actually appear.

`.dl` Motorola down-line load format (called "S-records"). Used to load over slow serial lines.

The three most often used options of `cc68` are `-o` *⟨name⟩*, which means the the output will be named *⟨name⟩*, `-c` which specifies a separately compiled module, and `-O` which causes the optimizer to be invoked.

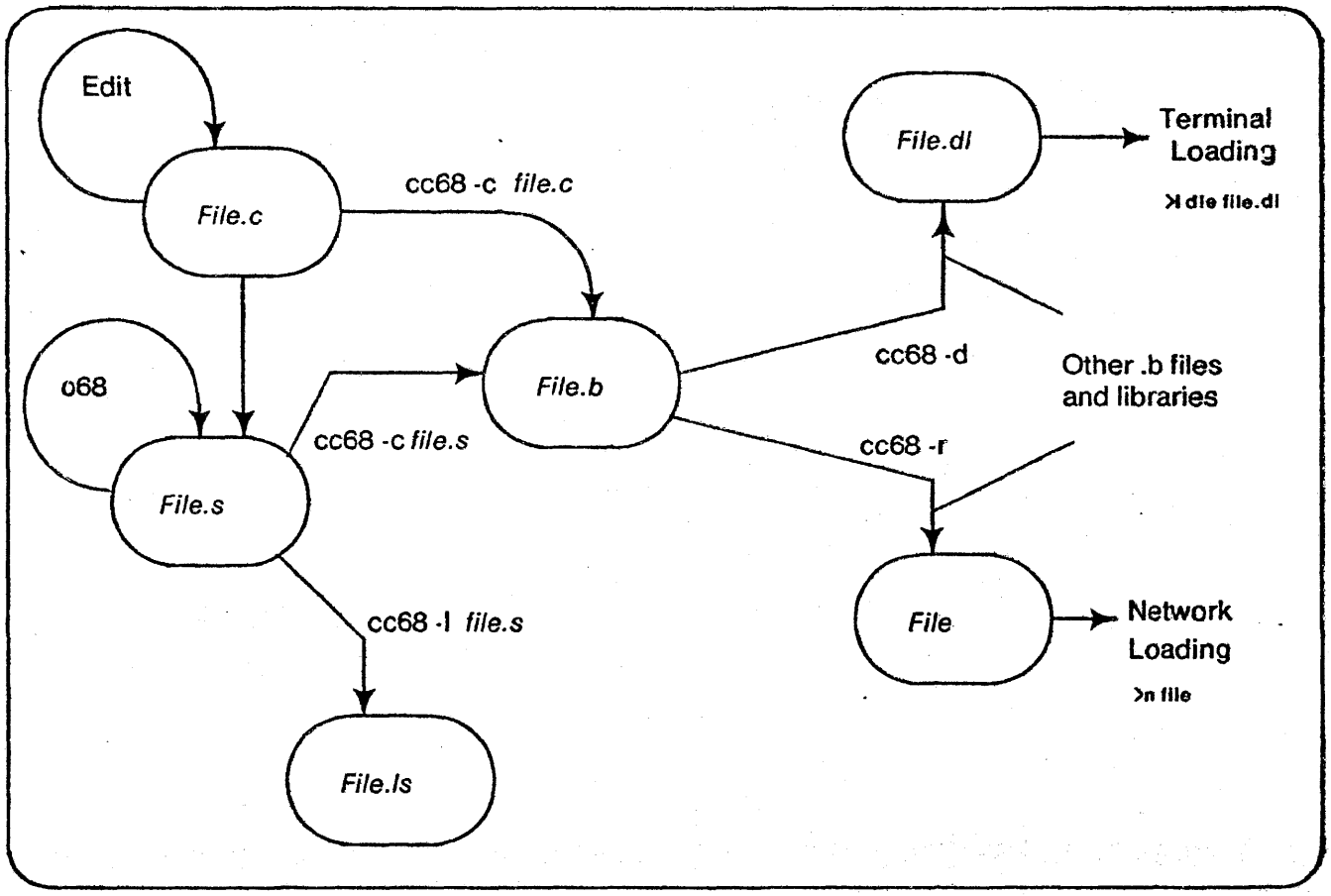

Figure 2-1: Developing Software with Unix

The `make` program simpifies the construction of programs using several modules [6]. Just create a file called `makefile` (or `Makefile`) under the directory with your programs to be compiled. The makefile consists of dependencies and commands. dependencies consist of a list of files to be made, followed by a colon, followed by a list of files upon which the others depend. Commands begin with a tab and end with a blank line. Thus, for example,

```
ethertip: enet.b tip.b
        cc68 -r -o ethertip enet.b tip.b

tip.b:  tip.c
        cc68 -O -c tip.c

enet.b: enet.h enet.c
        cc68 -O -c enet.c
```

is a makefile for a program called `ethertip` that consists of two modules, tip and enet. The first command line links the two modules together. The `tip.b` file depends on the corresponding source `tip.c`, and will be compiled with the optimizer because of the `-O`. The `enet.b` module is compiled from the corresponding `.c` file, but also includes a header (`.h`) file.

After creating the makefile, you can edit any or all of your sources, and just say `make` to remake the program. The `make` program reads the makefile, and infers the necessary commands from the write dates of the files. Continuing the above example, if we were to edit the `enet.h` file and perform a `make` command, the `enet.b` module would be recompiled and the program relinked with the following commands:

```
cc68 -O -c enet.c
cc68 -r -o tip enet.b tip.b
```

2.2.2. The 68000 Assembler

The 68000 assembler (`as68`) takes *file*`.s` and produces *file*`.b`, a non-text file containing an 8-word header and segments for text, data, symbols, data relocation commands, and text relocation commands. The header contains two words of magic numbers, the sizes of the text, initialized data, uninitialized data, symbol, and relocation-command segments, and the entry point (the start address of the top level routine). The listing might by slightly helpful in debugging; the values it gives for symbols however are incomplete. Uninitialized data symbols have no values in the listing, while all other values are relative to the start address, determined at load time.

Assembler options:

`-d`*<digit>*	Used for debugging the assembler only. The digit gives some indication of the level of debug printout.
`-e`	Only put external symbols into the output binary file.
`-g`	Treat any undefined symbols as globals. They must be eventually resolved by the loader.
`-L` *name*	Put the listing file into the file *name*. The `cc68` command normally puts the listing in a file with a name of the form *name*`.ls`.
`-l`	Produce a listing in *filename*`.list`, or other file if `-L` is given.

`-o` *name* Put the output in the file *name* instead of *filename*`.b`.

`-p` Print the listing on standard output.

`-s` Put the symbol table in the file `list.out`. The loader's symbol table is probably more useful, since it is made after relocation.

2.2.3. The 68000 Linker

The 68000 linker (`ld68`) produces a file called `b.out`, in a similar format to *file*`.b`. The primary tasks of the linker are to develop a symbol table for all of its inputs, and execute the relocation commands to produce absolute code. The output contains the same eight-word header as in *file*`.b`, text, data, and the symbol table.

Loader options:

`-D` *dsz* Set the size of the data segment to *dsz*. It is padded with zeros if too short.

`-d` Force the common segment to be defined (overrides second part of `-r`)

`-e` *ept* Make *ept* the entry point of the program. Otherwise the first address of the code segment is used.

`-l`*name* Search the library file named `/usr/sun/lib/lib`*name*`.a`. A library file is searched only as its name is encountered on the command line, so the order may be important. Note there is no space between the `-l` and the *name*.

`-M` Create a file listing the symbols and their values in `sym.out`.

`-o` *file* Output is named *file* instead of `b.out`.

`-r` Preserve relocation bits so that the output file can be used for input to `ld68` in a later run. Also prevents final symbol definitions for common symbols, and supresses "undefined symbol" diagnostics.

`-s` Strip the output file of all symbol table and relocation bits.

`-T` *addr* Start the code segment at *addr*.

`-u` *sym* Add *sym* to the symbol table. This is used when loading enitrely from a library, since you need at least one symbol reference to extract a "root" module.

`-v`*version* Specifies the version of the 68000 environment to assume. Currently the only available options are `-vm` for the Design Module, and the default for the SUN.

`-X` Discard locals starting with L (weaker than `-x`). This option is used by `cc68` to discard internally-generated symbols.

`-x` Discard all local symbols from the output file.

2.2.4. Producing Loadable Files

The binary produced by the linker is converted into a file suitable for down-line loading by most PROM monitors with the `d168` program. The only `d168` option is `-T` *st* which specifies *st* as the starting address The input file must be the first argument; options other than `-T` are ignored in silence. Symbol tables are preserved. See section 5.3 or 3.3 for information on actually loading the design module or SUN processor. The `rev68` program takes linked `b.out` files and does some final byte-reversing if the target machine has a different byte order than the current machine.

2.3. Header Files in /usr/sun/include

Several useful header files are located in the `/usr/sun/include` directory. This directory is searched automatically by the C preprocessor when you use `#include` directives.

amd9513.h Definitions for the AMD9513 timer chip. Jeff and several others have their own private versions of this file. Please complain to them about it, and the relationship with `timer.h`.

b.out.h Defines the format of a `b.out` binary file. Note that although you use the same header file, the bytes are reversed between the 68000 and the VAX, So you must run `rev68` to convert between the two.

buserr.h A structure definition matching the information pushed on the stack of the 68000 on a bus error, and the "function codes" as described in the MC68000 User's Manual [13].

chars.h Defines some mnemonics for control characters.

framebuf.h Definitions for the SUN Multibus frame buffer. See Chapter 4 for more information.

graphics.h Definitions for a "Raster-Op" graphics package which is frame-buffer independent. See Chapter 4, and the separate manual on this for more information.

graphmacs.h Some graphics macros for the SUN frame buffer. Most people will probably want to use the graphics package instead.

lisp.h A simple Lisp system defined in macros that map into C. Complain to Vaughan about this.

m68000.h MC68000 mnemonic definitions for registers, and some Macros for performing special instructions like setting the interrupt level from C.

m6821.h Motorola 6821 Peripheral Interface Accessory. This is a fancy name for the parallel I/O ports on the Design Module.

m6840.h Motorola 6840 Timer. This is the timer on the Design Module. It is so complicated

nobody here has ever figured out how to use it.

m6850.h	Motorola 6850 Asynchronous Communication Interface Accessory. This is the device that connects your terminal to the Design Module, and the Design Module to a computer. Almost everybody else calls it a UART.
m68enet.h	Definitions for the SUN Multibus Ethernet interface.
map.h	Vaughan's version of the definitions for the memory map on the 68000 processor board. Please complain to him about the duplication with `pcmap.h` and `noprotect.h`.
nec7201.h	Definitions for the NEC 7201 double UART. This is what connects your terminal on the SUN board, analgous to the Motorola 6850.
necuart.h	More stuff for the NEC 7201 UART, like the addresses and some utility macros. It is so complicated it takes two files!
noprotect.h	Yet another memory map file. This one defines the protection codes. Complain to Vaughan about the strange name.
pcmap.h	Jeff's version of the memory map definitions. Please complain to him about the duplication between this and `map.h` and `noprotect.h`.
reentrant.h	Defines a macro for interrupt handlers. Please complain to Vaughan about the strange name.
s2651.h	Definitions for the Signetics 2651 UART. This is the chip used on our octal UART boards.
statreg.h	Defines some symbols for the 68000 status register.
stdio.h	Some undocumented definitions for some kind of Unix-like standard I/O. Complain to Vaughan to document it.
sunemt.h	Defines the emt codes supported by the SUN Prom monitor.
sunmmap.h	Some macros for manipulating the old wire-wrap version of the memory map. Complain to Jeff to get rid of this.
sys.h	Some macros to do storage allocation of fixed size objects. Complain to Vaughan about the strange name.
timer.h	Some more definitions for the SUN processor board timer. See also amd9513.h.
vectors.h	Some symbols defined for the MC68000 interrupt and exception vectors.

2.4. Libraries

Some 68000 libraries are stored in the directory `/usr/sun/lib` in the form of archives compiled and assembled as .b files. The standard `ar` program, as described in the Unix manual, is used to manipulate these archive files. The `cc68` command normally searches the library `/usr/sun/lib/libc.a` automatically. Some of these functions are similar to the standard C library, but others are still under development. The sources should all be under `/usr/sun/src/lib` or `/usr/sun/src/libc`.

2.4.1. The C Library

In the interests of greater portability of low-level code between the design module and the SUN processor, some board independent I/O functions have been written and installed in `libc.a`. Any references to `putchar`, `printf`, and `getchar`, for example, will invoke functions to perform the I/O on the "console" terminal. Similarly `getenv` will behave rather like its Unix counterpart, with `getenv("TERM")` returning `"sun"`, etc.

There is also a set of lower-level functions in the standard C library. The device types are determined at link time, by supplying the appropriate set of routines, either as parameters to `cc68` (for example, `-vm`). All uart functions begin with 'line' and have the line number as their first argument. By convention, line number 0 is the local terminal or keyboard, and line number 1 is connceted to a remote host. This trades off size of the library for size of the calling code and a little speed.

`char lineget(`*line*`)`
: Get a character from line *line*, assuming the receiver is ready.

`lineput(`*line*`, `*chr*`)`
: Put character *chr* on line *line*, assuming the transitter is ready.

`int linereadytx(`*line*`)`
: True if the transmitter is ready on line *line* to accept another character to be transmitted.

`int linereadyrx(`*line*`)`
: True if the receiver is ready on line *line*.

`lineservice(`*proc()*`)`
: Set the given procedure pointer to be the interrupt service routine for both lines. *proc* should be declared with the `reentrant()` macro, described in section 2.3.

`linearmrx(`*line*`)`
: Enable receiver interrupts on the given *line*. The interrupt service routine should already be set up.

`linedisarmrx(`*line*`)`

Disarm receiver interrupt on the given *line*.

`linearmtx(`*line*`)`
Arm transmitter interrupt for line *line*.

`linedisarmtx(`*line*`)`
disarm transmitter interrupt for line *line*.

`lineresettxint(`*line*`)`
Reset a transmitter interrupt, when there are no more characters to print.

`linereset(`*line*`)`
Reset *line*. Default to interrupts disarmed.

It is the responsibility of the individual libraries to deal with the problem of independently setting and clearing control bits for arm and disarm. The user need not keep track of the bits explicitly.

The meaning of `lineready==0` is that the line is busy for whatever reason, whether no carrier, no DTR, no CTS, or reception/transmission proceeding. It is assumed that all operations will be with 8 data bits, 2 stop bits, and no parity bit.

No provision is made for detecting UART errors. It may be reasonable to attempt low-level error correction/detection in lineget and possibly lineput; however this should not be considered a substitute for higher level error correction/detection (checksums on downloading, perhaps use of Dialnet).

2.4.2. The PUP and Leaf Libraries

The PUP [5] library is stored in `/usr/sun/lib/libpup.a`. Since these routines are all described in section 9 of the Unix manual, (available online with the `man 9` command), they are not described in detail here. `cc68` will search this library if it is given the `-lpup` option.

The `-lleaf` option on a `cc68` will search the leaf library, stored in `/usr/sun/lib/libleaf.a`. Leaf is a remote file access protocol, based on the Sequin reliable packet stream protocol. There are Leaf servers running on most large timesharing machines. Documentation of the library is in `/usr/local/doc/leaf/LeafUser.press@Shasta`.

2.4.3. The SUNOS Library

The SUNOS library is stored in `/usr/sun/lib/libsunos.a`, and can be searched with the `-lsunos` option of `cc68`. The SUNOS library is discussed in chapter 7.

2.5. The 68000 C and Pascal Calling Sequence

The stack grows downward, towards smaller addresses. Two address registers are used to access the stack, a6 (the frame pointer) and a7 (the stack pointer). The stack pointer is the standard one for the 68000, in that BSR (Branch to SubRoutine), JSR (Jump to SubRoutine), and PEA (Push Effective Address) all use it. Note that exceptions do not default to it, but rather to the System Stack Pointer, a register not accessible in user state.

The code produced by the C and Pascal compilers result in a6 serving as a pointer to a linked list of stack frames. A stack frame is a region of the stack associated with the calling or activation of a function. Stack frames are stored contiguously on the stack. Stack frames have five components; from low addresses (top of stack) to high they are:

1. Locals. These may be of any size, and will occur in the reverse of their declaration order (or, perhaps more mnemonically, in their declaration order going away from the Frame Center). They will be contiguous to within the word alignment restriction, i.e. non-chars will be aligned at even addresses. If the first local is char its address is the same as though it were a short (a quirk of the compiler). The compiler also allows one additional byte beyond the last local.

2. Registers. This region contains some subset of the registers d2-d7 and a2-a5 saved on entry. The registers saved are those actually used. The convention is made that C subroutines always preserve these registers, as well as a6 and a7, but change d0,d1,a0 and a1 unpredictably. The optimization is such that this convention holds down to the statement level (or even lower!). Alignment is on multiples of 4. Because of the additional byte allowed by the compiler for locals, that byte and up to 3 more (the worst case being when the locals actually declared end on a multiple of 4) are all unused.

3. Frame Center. This contains a 4-byte pointer to the Frame Center of the next stack frame down the stack (i.e. at a higher address).

4. Return Address. This contains the caller's 4-byte return address.

5. Arguments. The arguments passed by the caller are implicitly cast as ints before being pushed on the stack. Hence they are stored in consecutive 4-bytes regardless of their actual size. They are stored in calling order, with the first argument closest to the Frame Center. It follows that the order in which they are pushed is the reverse of the order they are written.

2.6. The "S-Record" Down-Line Load File Format

A **.dl** file consists of a series of records each having seven components:

1. The letter S.
2. A type, a digit in the range 0 to 9.
3. A two digit (one byte) count, giving the number of bytes in the record.
4. An address, either 16-bit (two hex digits) or 24-bit (three hex digits).
5. *n-3* or *n-4* bytes of data, depending on the address type, where *n* is the count given in 3.
6. A one-byte checksum. The checksum test is that the sum of the bytes in items 3 through 6 must be congruent to 255 mod 256, i.e. must have 0xFF in the least significant byte.
7. The end of the line.

The types are as follows:

Code	Use	Features
0	Header	Ignored by MACSbug, and no longer generated
1	Data	Two-byte address, bytes in hex (not used)
2	Data	Three-byte address, bytes in hex
8	Trailer	Three-byte address, bytes in hex
9	Trailer	Two-byte address, bytes in hex (not used)

If a header is given it goes at the start. A trailer must appear, and goes at the end. The rest of the file consists of data records. The header is currently ignored. Each data record is loaded into memory starting with the address specified in in the record, provided it passes the checksum test. The trailer serves two functions: to terminate reading, and to load PC with the trailer's address, giving a mechanism for defining the entry point of a program.

3. Using the SUN Processor

The SUN processor is a powerful single board computer containing a Motorola MC68000 CPU, memory with management and parity, and some I/O devices. The board plan is illustrated in figure 3-1.

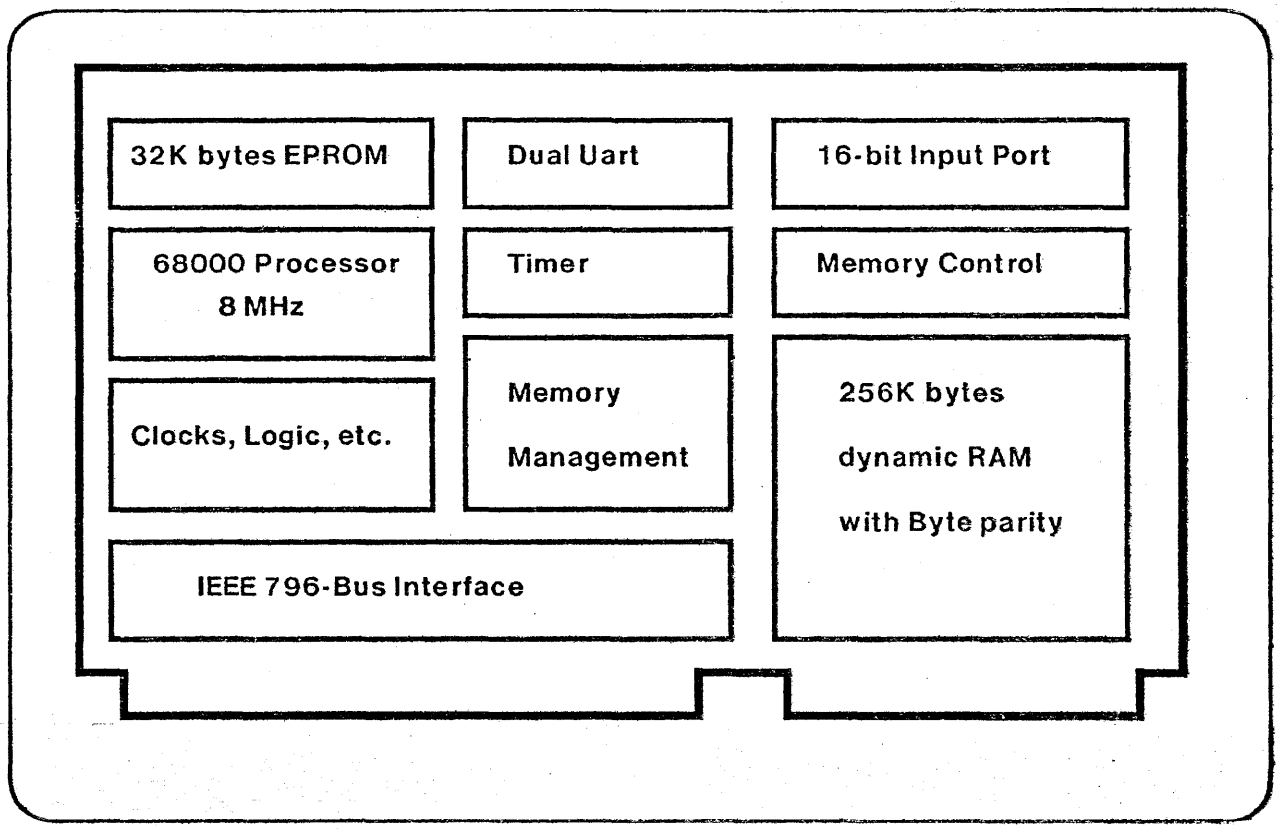

Figure 3-1: Layout of the SUN Processor Board

3.1. Introduction to the SUN ROM Monitor

Most use of the SUN processor involves interacting, at least initially, with the ROM-resident monitor. The following sections discusses the purpose of the monitor, and how to use it. Note: the discussion below assumes (occasionally) the use of the PC-board version of the Sun Processor, which has a two-level memory map. Users of the Versions 1 and 2 wire-wrap boards can probably deduce the points of difference; however, neither of these boards will be supported in the future.

Although the primary function of the ROM monitor is to provide a simple console for the workstation, there are a few features which affect the user programs that run under it. For simple programs, especially those using standard I/O routines, the characteristics of the monitor should not be important. However, if a program makes direct use of interrupts or I/O devices, a few critical details are relevant.

3.1.1. What is the monitor?

It is useful at this point to give a brief description of the operation of the monitor (SunMon), mostly to provide a context for understanding the few rules imposed upon user programs.

The monitor has four major functions: initialization on processor reset, memory refresh, providing Emulator Trap service (see section 3.8), and "intelligent console" facilities. Although the last may be the most visible, the first two are the most important; the processor would be essentially unusable without them.

When the processor is reset (either when the "Reset" switch is hit, or when the power first comes on), SunMon gains control. It initializes the on-board I/O devices (timers and UARTs), sizes memory, sets up the Segment Table and Page Table, initializes the parity state of the on-board RAM, creates the RAM refresh routine, and initializes the interrupt and exception vectors. After this, control is transferred to a module that manages the "console" functions.

Memory refresh is done by the processor because it actually does not cost anything in terms of performance, and because it greatly simplifies the hardware design. The memory is refreshed by simply reading 128 consecutive words every 2 milliseconds (some memory chips may need slightly different refresh rates). This is done by executing a routine consisting mostly of NOPs. This routine is stored in RAM, and so a malfunctioning program may damage it and thus cause havoc (since the contents of memory will be lost).

The console functions are implemented with fairly straightforward routines that communicate with the user via the two on-board UARTs. If a Frame Buffer is available, the monitor will use it for output instead of the console UART (command are still entered via a keyboard connected to the UART.)

In versions of SunMon that support ethernet bootstrapping, the console also uses the Multibus Ethernet interface. In any case, all I/O is done using "busy-waits", and the code runs at the highest interrupt priority. Therefore, if a user program is interrupted with the "Break" key on the console terminal or with some other exception, the monitor will run correctly unless its global data area has been damaged. Also, if the user program is then continued, it should be unaffected by the interruption save for the possible loss of some I/O.

3.1.2. Absolute Rules

From the preceeding section, it should be fairly obvious that one major rule is necessary to prevent a monitor crash: do not trash the refresh routine or monitor globals. In general, the first two pages of memory are reserved for the monitor and should never be written by user programs; however, user code may want to change exception vectors occasionally. It is legal to change any exception vector, *except* the "Level 7

Autovector", at 0x7C (used for refresh timing), and *any* "User Interrupt Vector", between 0x100 and 0x3FF, inclusive. The refresh routine and monitor globals live in the region reserved for "User Interrupt Vectors", because the Sun processor board does not support their use.

Certain other exception vectors (for example, the vector for a Breakpoint trap) are used by the monitor. However, it may be possible to alter these without dire results. Any program altering the refresh routine and interrupt vector must take responsibility for doing proper memory refresh.

One other rule is important: user programs should not modify the Context Register directly, but should use the facilities described in section 3.8.

3.2. Getting Started

3.2.1. Initializing the Workstation

The first step in using the SUN processor is making sure it is turned on. There is probably a switch labeled "ON" and "OFF". It will work better if the power switch is in the "ON" position. (Version 2 processors have a "power" light on their control panels.) A terminal should be connected to the communications cable extending from the processor. If you have a serial line to a host computer available, it should also be connected to the communications cable. Finally, if an ethernet interface is present, it should be connected to an ethernet transceiver cable.

There is one switch, labeled "Reset" (it may actually be a power on/off switch.) Pressing and releasing the Reset switch should be done after initial power-up of the machine, and whenever you want to really reset everything. After a few seconds, the monitor should identify itself on the console terminal, with a message looking like

```
Sun Network Monitor, Version 0.9 - 0x20000 bytes of memory
```

The word "Network" may be omitted; certain commands pertaining to the ethernet only work with network monitors. If this message does not appear, and the "halt" light is not lit, check the terminal's status and connection. If the halt light is lit, and if repeated use of the Reset switch has no effect, your hardware may be broken. The Reset operation will probably destroy the contents of memory.

Pressing and releasing the "Break" key on the console terminal switch causes a trap (also known as an "Abort") to the monitor so that debugging commands may be given. You may continue an aborted program; see the C command, described in section 3.3.

3.2.2. Some Sample Programs

The following are some useful programs available in the default bootfile directory. Most of their sources are in appropriate subdirectories of `/usr/sun`. They all can be loaded and started with the `n` *name* command of the monitor. **[Note: the locations of source files here are all wrong!]**

`cyl4` A short program that displays four circles that move diagonally on the screen. A good example of simple animation. Runs only with a frame buffer, of course.

`edp` A simple ethernet diagnostic program. Source is in `/usr/sun/diag/edp`. You might have to load the `.dl` version, especially if your ethernet interface is not working.

`kal` A kalidescope program, originally written in BCPL for the Alto, and transcribed into C. Really impressive graphics demo for the frame buffer. Source is in `/usr/sun/demo`. Also see `ikal`, an interactive version.

`memtest` A memory diagnostic, sources in `/usr/sun/diag/memtest`. When your workstation has nothing better to do, it could run this to help check for faulty memory.

`monhelp` As described below, this is a file which prints out a quick summary of the available monitor commands. *Note: various versions of* `monhelp` *exist, corresponding to the various versions of the monitor. On version 0.7 and later monitors,* `monhelp2` *is the program to run. Please bug Jeff to clean this up!*

`rect` Another graphics demo for the frame buffer, The sources are located in `/usr/sun/demo`. This just does some "random" raster-ops in rectangles on the screen. `frect` is an impressively fast version of `rect`.

`sunbfd` A program which lists the "standard" bootfiles available via the ethernet bootloader.

`suntty` A program which simulates a multi-window terminal on the frame buffer, with a separate PUP telnet connection in each window. The source to this and the `tty` program is in `/usr/sun/tty`.

`tip` A multi-user PUP telnet program. If you have any octal UART boards in your machine they may each have a PUP telnet connection, as well as the primary terminal.

`tty` Another version of the multi-window terminal program. This one uses the serial line instead of the ethernet interface, so you talk directly to one host.

3.2.3. A Simple Example

We will now step through how to write, compile, load, and run a simple program. First, we use our favorit editor to enter the following program into the file `/mnt/smith/test.c`:

```
main()
 {
    /*
     * A simple tst program for the SUN workstation
     */

     printf( "Hello world!\n" );
 }
```

We now compile the program with the following command:

```
cc68 -r -o test test.c
```

We then go to our workstaion and type the command to load and run:

```
>n /mnt/smith/test
Hello world!
>
```

3.3. The ROM Monitor Commands

The command format understood by the monitor is quite simple. It is:

<verb><space>[<argument><return>]*

The *<verb>* part is always one alphabetic character; case does not matter. *<space>** means that any number of spaces is skipped here. *<argument>* is normally a hexadecimal number or a single letter; again, case does not matter. As indicated by *[]*, the argument portion may be optional. When typing commands, *<backspace>* and *<delete>* (also called *<rubout>*) erase one character, control-U erases the entire line.

The commands are:

A *n*	"Open" A-register *n* ($0 < n < 6$). See the discussion below of "open".
B	Set a breakpoint. You will be prompted with the old breakpoint address; give the new address at which you want a breakpoint trap instruction inserted.
C *addr*	Continue a program. The address *addr*, if given, is the address at which execution will begin.
D *n*	"Open" D-register *n* ($0 < n < 7$).
E *addr*	"Open" the word at memory address *addr*; odd addresses are rounded down.
F *Bootfile name*	Load, but do not start, a file via the Ethernet. The program can be started with the **G** command. Normally, the current PC is set to the entry point of the loaded program. Of course, this command is only available on network monitors.
G *addr*	Start the program by executing a subroutine call to the address *addr* if given, or else to the current PC.
H	a minimal amount of help will be given; just a short list of commands. On network

monitors, this actually involves boot-loading a help program via the ethernet. If the ethernet interface is not working, no help is available.

`I` *mode* — set UART operation *mode*: 'A' means your terminal talks to the monitor, 'B' means your host computer talks to the monitor (not very useful if directly invoked), and 'T' means that you talk to the computer ("Transparent" mode) until you hit the transparent mode escape character (initially set to control/shift/six or control/up-arrow) followed by a "c". 'S' toggles the use of the Frame Buffer as the console output device; i.e., it selects it if it is not being used, and selects console UART otherwise. The 'S' option has no effect if there is no Frame Buffer present.

`K` — "Soft Reset": resets the monitor stack and the default escape character. Useful after exceptions or other anomalous situations. This may confuse the monitor if a breakpoint trap is set.

`L` *Host-command* — This sends *Host-command* to the host computer, does an implicit `I B`, and sends a `\` to the computer to indicate that it is ready to be downloaded. The *Host-command* is normally `dlx file.dl`, which will put the terminal back into normal mode when the file is downloaded.

`M` *m* — On the SUN-1 processor, "open" Map register *m*. On the SUN-2 processor, "open" Segment Map register *m*.

`N` *Bootfile-name* — Load and start a file via the Ethernet. *Bootfile-name* is the pathname of the file to be bootstrapped. It should be in `.r` format, produced with the `-r` option of `cc68` or the `rev68` program. This command is only available on network monitors.

`O` *addr* — "Opens" the byte location specified. The byte vs. word distinction is a problem on the Multibus, since the convention on byte ordering within words is different for Multibus addresses.

`P` *p* — On the SUN-2 processor, "open" Page Map register *p*. On the SUN-1 processor, this is used to set the PID (Process Identification) register to *p*.

`R` — "Opens" the miscellaneous registers (in order) `SS` (Supervisor Stack Pointer), `US` (User Stack Pointer), `SR` (Status Register), and `PC` (Program counter). `SS` may not be altered.

`S` *S-record* — This causes the monitor to accept the *S-record*. Normally done by the host computer in `L` mode, this responds with a two-digit record count and one of `L` for length error, `K` for checksum error, or `Y` for success.

`X` *char* — set the transparent mode escape character to *char*. Because of the way that the parser treats spaces, the escape character cannot be set to be a space.

"Opening" a memory word, map register, or processor register means that the address or register name is displayed along with it current contents. You may then type a new hexadecimal value, or simply `<return>` to go on the next address or register. Typing `Q` will get you back to command level. For registers, "next"

means within the sequence D0-D7, A0-A6, SS, US, SR, PC. For example, the following commands set location 1234 to 5678, and register D1 to 0F00. The user types the underlined parts, with a return a the end of each command.

```
>e 1234
001234: 23CF? 5678
001236: 0000? q
>d
D0: 00000001?
D1: 00000231? 0f00
D2: 01203405? q
>
```

3.4. Loading Programs

One of the primary uses of the monitor is to load programs into the processor's memory. Programs can either be loaded via a serial line connected to a host computer, refered to as "down-line loading", or via the ethernet, refered to as "net-loading". In usual terminology, "down-line loading" often refers to any method loading one computer from another, but it is useful to make the distinction here. Net-loading is usually much faster, but both modes have their advantages and disadvantages.

3.4.1. Down-line Loading

Down-line loading involves transferring a program file over a serial line. The file must be converted into a format known as "S-records" before transmission, either using the `d168` command or the `-d` flag of `cc68`. Files in this format usually have a `.dl` extension.

Suppose the file we want to load is called `test.dl`. Assuming that you have used "transparent" mode to log into the host computer and have set your working directory properly, you should then "escape" from transparent mode. Then, issue the command

```
L dlx test.dl
```

This will transmit the command `dlx test.dl` to the host, and then cause the monitor to accept future commands from the host. If all goes well, you should see a string of periods on your terminal, and then a monitor prompt when the load is done. You may then start your program with the `G` command; normally, the current PC is set by the downloader to be the entry point of the program.

If the new monitor prompt comes immediately, this means that the `dlx` program detected an error, and your program could not be loaded (probably because it could not be read). If the periods stop coming (one should print every few seconds), this means that the loader has hung. You should hit Reset or the "Break" key, change to transparent mode, and type control-C (or your normal interrupt character) to abort the `dlx` command. You may also have to issue the Unix `reset` command to put your terminal line back into a normal mode.

Attempting to down-line load a file not in S-record format will probably cause strange behaviour, although the `dlx` program attempts to detect this error. Also, you may omit the `.dl` extension in most cases, i.e.,

```
L dlx test
```

should be equivalent to

```
l dlx test.dl
```

unless both `test.dl` and `test` are S-record files.

3.4.2. Net-loading

If you have a network monitor, and an ethernet interface, you may load programs over the ethernet. Program files to be loaded in this way should be in "reversed `b.out` format"; this means that a file produced by the loader should be converted using the `rev68` command, or the `-r` flag of `cc68` should be used in compilation. Files in this format often have a `.r` extension, but sometimes they have either a `.Boot` extension, or no extension at all.

Suppose you want to load the file `/mnt/person/test.r`. You should give the command

```
F /mnt/person/test.r
```

You should almost immediately get a new monitor prompt. You may then start your program with the `G` command; normally, the current PC is set by the net-loader to be the entry point of the program. Alternatively, you may give the "load-and-go" command

```
N /mnt/person/test.r
```

which will load the file and immediately start it.

If the net-loader fails to load the file, it will print a period and try again, up to a reasonable limit. If it gives up, it will print `Timeout` and return to the monitor. Even if a file is not successfully loaded, it is quite probable that the memory has been altered.

If you get an exception when net-loading a file, it may be because there is a hardware problem, but it is more likely to be because you loaded something not in reversed b.out format. Of course, if the exception occurs with the load-and-go (`N`) command, it may have been caused by your program.

The network bootstrap server running under Unix interprets filenames not beginning with '`/`' as relative to `/usr/sun/bootfile`. Thus,

```
N memtest
```

will load the file `/usr/sun/bootfile/memtest`. In fact, the `H` (help) command in the network monitor is equivalent to doing an `N monhelp`.

Normally, bootload requests are broadcast to all servers on the net. However, you can specify the name of

the host you want to boot the file from by preceeding the filename with the hostname and a colon. For example,

```
N shasta:memtest
```

will get the program "memtest" from the PUP host named "shasta".

Note that it is *not* possible to load more than one file at a time with the net-loader. If you want to load more than one file (presumably into different areas of memory), you must use the down-line loader.

3.5. Memory Mapping

The SUN processor is provided with a map so that you can map pages of 2K bytes anywhere in your address space. The structure of the virtual address is given in figure 3-2. SunMon, during initialization, sets up the Segment Table and Page Table in a "standard" way which makes all memory and I/O devices available to user programs. User programs may change these maps (although page 0 and whatever pages likely to contain the top of the Supervisor stack should not be remapped, or memory refresh may fail). However, for simple programs the initial mapping may be best left alone.

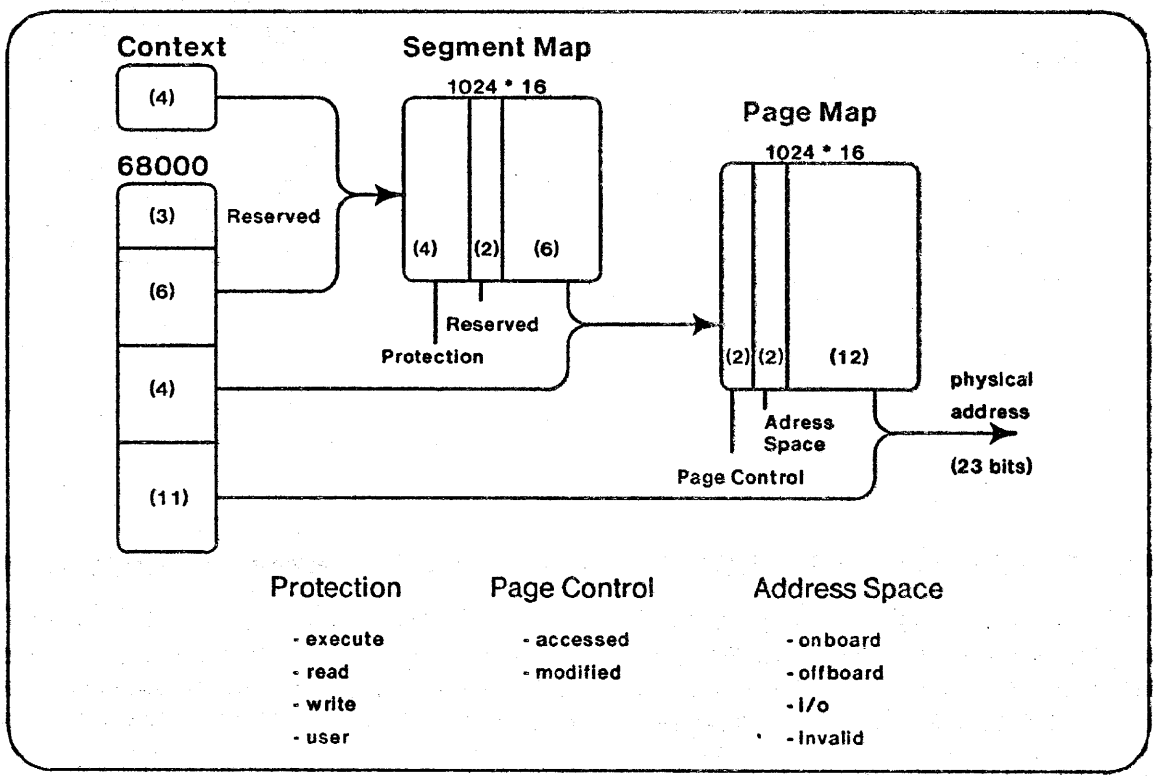

Figure 3-2: Memory Mapping on the SUN Processor

Each actually existing page of on-board RAM is initially mapped so that its physical and virtual addresses

are identical. This means that each segment, starting at segment 0, is fully mapped (up to the limit of available memory). Segments are only initialized for context 0. Segment protection is set so that both Supervisor and User modes have Read, Write, and Execute access to every segment.

Two other physical address spaces are mapped into the memory address space. The first 64K bytes of MultiBus I/O space is mapped at the top of the virtual address space, and extends for 32 pages (64K bytes). Thus addresses from 1F0000 to 1FFFFF get mapped to the Multibus I/O space. The Ethernet interface and most commercially available Multibus I/O devices use this space.

The rest of the high megabyte of mappable address space is mapped as MultiBus memory. Address from 100000 to 1EFFFF are mapped to Multibus memory space addresses 0 to 0EFFFF, respectively. This is where the frame buffer resides.

The physical address space (the 24 bit addresses used in the internal bus) is divided into eight parts, as described below.

0 - 1FFFFF	Mapped address space, as described above. There is usually 128K to 256K bytes of on-board RAM, with a limit of 512K. This space can also be mapped into the Multibus I/O or Multibus Memory space.
200000 - 3FFFFF	On board PROM0. See the discussion below on "boot state".
400000 - 5FFFFF	On board PROM1.
600000 - 7FFFFF	The on-board double UART. Channel A data register is at 600000, command register at 600002, Channel B data is at 600004, and B command is at 600006.
800000 - 9FFFFF	On board Timer chip. 800000 is the Data register, 800002 is the Command register.
A00000 - BFFFFF	Page map in SUN-2, PID register in SUN-1.
C00000 - DFFFFF	Segment map in SUN-2, page map in SUN-1.
E00000 - FFFFFF	Context register in SUN-2.

In "boot state", the state of the system after reset, read and execute accesses to any location 0xxxxx in mapped addresss space are redirected to come from the corresponding location 2xxxxx (in the PROM0 address space), but write accesses to the mapped address space go to on board RAM. Also, all interrupts (including normally "non-maskable" ones) are inhibited. In this way it is possible to initialize RAM just after reset. Boot state is exited on the SUN-1 processor by writing to 0xE00000, and by writing to the PROM0 address space on the SUN-2 processor.

When the monitor is initialized, it sets the Supervisor Stack Pointer to 0x1000, and the User Stack Pointer to the top of available memory. User programs may change these registers, providing that they do not cause the supervisor stack to overflow into unmapped address space.

More exact detail on the memory mapping, as well as constant definitions useful for C programs, may be found in `/usr/sun/include/sunmmap.h`. *(Note:* `sunmmap.h` *currently describes only Version 1 processors.)*

3.6. Traps

The monitor initializes the trap vectors so that it gets control of any exception or interrupt. Some, such as the memory refresh timer interrupt, are handled internally. Others have special meanings (for example, the "trap #1" operation is treated as a breakpoint trap). For exceptions or interrupts not internally handled, the monitor will print a message such as `Exception: Tr` and then return to command level.

The messages printed use a two-letter code; here is a list of these codes and their meanings.

`II`	Illegal Instruction: an illegal instruction code was executed
`ZD`	Zero Divide: division by zero
`Ch`	Check: a CHK instruction faulted
`TV`	TRAPV: a TRAPV (trap on overflow) was taken
`Pr`	Privilege violation: attempt made to execute privileged instruction while in user state
`U0`	Unimplemented 0: an opcode 1010 was executed (emulator trap)
`U1`	Unimplemented 1: an opcode 1111 was executed (emulator trap)
`Un`	Unassigned: trap was made to unassigned vector.
`L1, L2, L3, L4, L5, L6`	Interrupt Autovector: an Autovectored interrupt was taken at one of levels 1 through 6.
`Tr`	Trap: a trap instruction was executed.

Several exceptions are handled specially by the monitor. A breakpoint trap (instruction "trap #1") causes the message

```
Break at pc
```

to appear. A trace trap evokes the message

```
Trace trap at pc
```

to appear. Use of the "Break" key causes

```
Abort at pc
```

to appear. In each case, the *pc* shown is that of the next instruction to be executed. For further information on the use of these three traps, see section 3.7.

A Bus Error trap (usually caused by attempting to access non-existant memory or devices) gives the message

```
Bus Error: address access-address at pc
```

Similarly, an Address Error trap (usually caused by attempting to access a word with an odd address) causes the monitor to print

```
Address Error: address access-address at pc
```

In either case, the *access-address* is useful in helping to determine the cause of the trap. It is possible to continue from these traps, although the apparent effect of the faulting instruction is not always defined.

"User Interrupt Vectors", locations 0x100 through 0x3FF, are not available for use as such on the SUN board due to the hardware design. For this reason, this area of memory is used by the ROM monitor for storing globals and the RAM refresh routine.

3.7. Tracing programs

The monitor provides several facilities for tracing program execution. They are quite primitive, however, and basically require you to understand your program at the machine code level. However, if you have a symbol table listing of your program (created using `nm68`), you will be able to at least know where each routine starts.

3.7.1. Breakpoint traps

The use of a Breakpoint trap (BPT) allows to run a program and regain control when execution reaches a certain location. The monitor currently can only maintain one breakpoint trap at a time. A breakpoint trap is set using the `B` command; after giving this command, you will be given the address of the previous BPT and prompted for a new address. For example,

```
Break 001000?
```

means that a BPT is already set at location 1000. At this point, you could type a 0 to clear the BPT, a return to leave things as they are, or a new address at which to set a trap (the old trap will be cleared).

If you had gained control of your program before setting the trap by using the "Break" key, you might want to continue it using the `C` command. Otherwise, you will probably want to start the program using the `G`

command. Execution will then proceed until the trap is reached, at which point you will get a message such as

```
Break at 001000
```

At this point, you may examine the location at which you set the BPT and you will find that it contains the original instruction. You may clear the BPT or set a new one at this point. If you do not, you may continue using the C command, which will execute the "broken" instruction, then reset the BPT and continue. If you give an address to the C command, the breakpoint trap will not be reset, unless for some reason you take a Trace trap.

If you load a new program while a BPT is set, the monitor will normally be able to detect this. On the other hand, if you give the K command ("Soft Reset") while a BPT is set, and then set a new one, wierd things will happen if the first trap is taken. *Jeff should fix this!*

3.7.2. Trace traps

The support for Trace traps (single-stepping a user program) is even more minimal than the support for Breakpoint traps. To set a trace trap, you should use the R command, proceed to the Status Register (SR), and alter it so as to inclusive-OR it with 0x8000. Similarly, the trace trap can be cleared by ANDing the value of SR with 0x7FFF.

Once the trace bit is set in the SR, you should then give the C command to continue the program (the G command cannot be used in this way); to start a program with the trace bit set, give the command C *starting-address*. Subsequent steps may be made by using the C command without an argument.

For complex reasons, it is not possible to single-step after a Breakpoint trap is taken, unless you first clear the BPT. Once you have stepped one instruction, you may then reset the BPT. *Jeff may fix this sooner or later.*

3.8. Emulator Traps

The ROM monitor is able to provide several services to user programs via "Emulator Traps" (EMTs). An EMT is a convenient way of entering the monitor which does not depend directly on the absolute addresses used. Instead of executing a `jbsr` instruction, a program wishing to use the emulator first pushes a "trap type code" on the stack, and then executes a `trap #15.` instruction. In most other respects the operation is identical to a function call.

The services provided by the EMT facility fall into three categories: information, I/O, and memory-management. Some of these, such as the memory-management operations, are restricted to supervisor mode. The following section gives the C-language calling sequences and descriptions for the EMTs; assembly-

language definitions are available in the file `/usr/sun/include/sunemt.h`.

In general, if one of these functions encounters an error condition, it will return the value -1. In particular, attempting to execute a trap reserved to supervisor mode while operating in user mode will result in an error return.

3.8.1. Information EMTs

`int emt← ticks()`

Returns the number of milliseconds since the monitor was last booted. This is incremented whenever memory is refreshed, at least every 4 milliseconds. The accuracy is sufficient for time-of-day uses, if the crystal on the processor card is working right.

`int emt← getmemsize()`

Returns the size of the on-board RAM in bytes.

`int emt← version()`

Returns the "version" of the ROM monitor; the most significant byte is the major version number, the next byte is the minor version number. For example, `0x0105` corresponds to version 1.5.

3.8.2. I/O EMTs

```
emt← putchar(c)
char c;
```

Prints the specified character on the Console. If `c` is a linefeed or a carriage return, then it is followed by a carriage return or a linefeed, respectively. The I/O is done using busy-waiting.

`char emt← getchar`

Returns the next character typed on the Console keyboard. Normally, the character is also echoed on the Console. The I/O is done using busy-waiting.

```
setecho(flag)
int flag;
```

Controls whether characters input from the Console by `emt← getchar()` are echoed or not; they are echoed if and only if `flag` is true.

3.8.3. Memory Management EMTs

These EMTs are provided for use by the kernel of an operating system, and are restricted to supervisor mode only. They are necessary because it is not possible to access the segment registers for one context while running in another context. Since there may be no properly initialized segments in a given context, it is not

possible for the kernel to simply switch contexts before changing the segment map; this operation must be done by code running entirely in ROM and processor registers. The memory management EMT's provide this service. Note that it is *essential* that the context register not be changed except via the EMT described below.

```
short emt← getsegmap(cxt,segno)
int cxt;
int segno;
```

Returns the contents of segment map entry number `segno` in context number `cxt`.

```
emt← setsegmap(cxt,segno,entry)
int cxt;
int segno;
short entry;
```

Sets segment map entry number `segno` in context number `cxt` to `entry`.

```
int emt← getcontext()
```

Returns the current value of the context register.

```
emt← setcontext(cxt)
int cxt;
```

Sets the context register to `cxt`.

Here is an example of the use of the `emt← ticks` global to derive an accurate timer, counting seconds. *(Warning: a previous version of this example contained a program logic error that rendered it inaccurate!)*

```
main()
{
        long NextTick;          /* value of RefrCnt at next second */
        long ThisTick;          /* temporary; used to avoid a race */
        long seconds = 0;

        ThisTick = emtticks(); /* initialize loop invariant */

        for (;;) {
                printf("Time is %d\n",seconds++);
                NextTick = ThisTick + 1000;     /* predict next second */
                while ( ThisTick < NextTick )   /* busy-wait */
                        ThisTick = emt_ticks();
        }
}
```

4. The SUN Graphics System

The SUN graphics system is a high-resolution bit-mapped frame buffer on one Multibus board. The general organization of the graphics board is illustrated in Figure 4-1. There is only a small amount of hardware assistance to perform the simple high bandwidth operations (called "RasterOps"). This results in a simple, yet flexible graphics device, with high enough performance for sophisticated user interfaces.

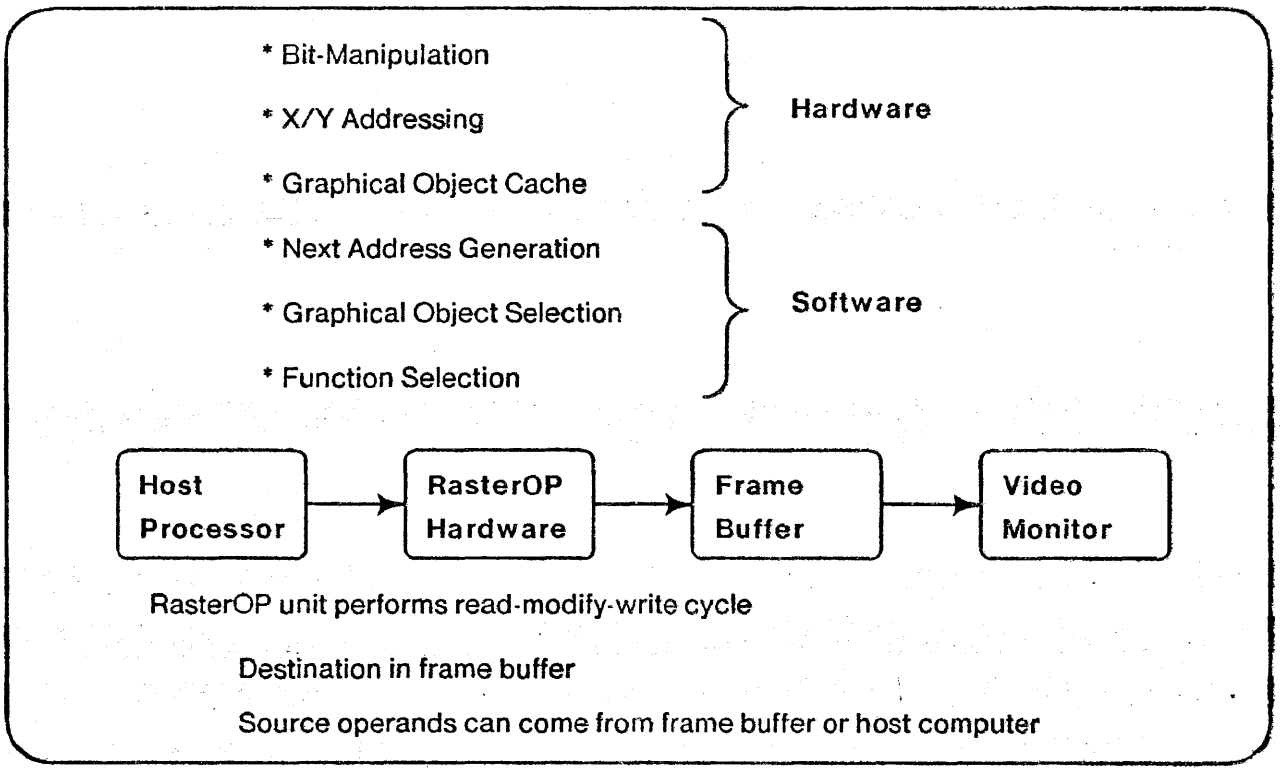

Figure 4-1: The SUN Graphics Board

4.1. Graphics on the SUN workstation

As shown in Figure 4-2, the nominal viewable area of the screen is 1024 pixels high and 800 pixels wide in "portrait" mode (similar to the Alto display). Other configurations (like the Xerox Large Format Display) are also possible with appropriate changes to PROMs on the graphics board. The Large Format Display is 808 pixels high and 1024 pixels wide. This display is compatible with the display used in the Xerox "Star" 8000 and "Dolphin" 1100 workstations. The points are addressed by X and Y (column and row), starting with (0,0) in the upper left corner of the screen. From one to sixteen consecutive pixels may be read from or written to the frame buffer in one memory cycle (one microsecond).

Figure 4-3 illustrates the concept of "RasterOp", as developed by Newman and Sproull [14]. A RasterOp

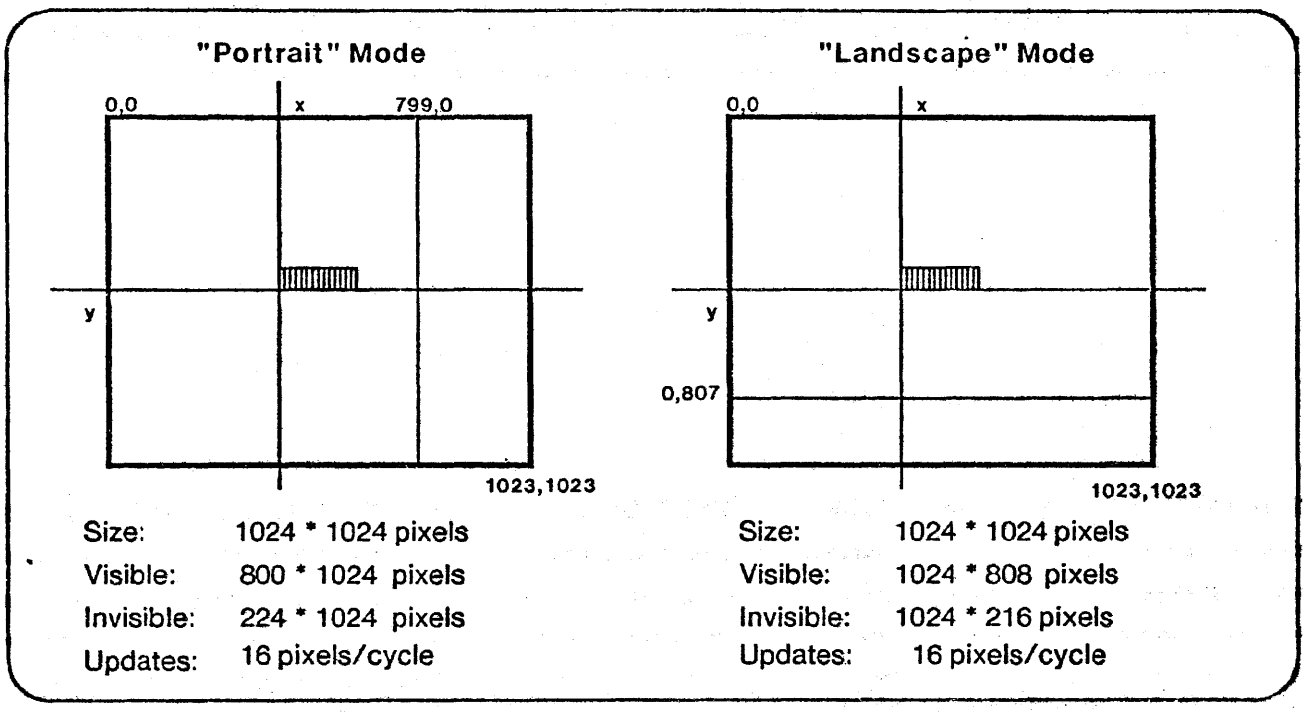

Figure 4-2: The SUN Graphics Screen

sets a destination rectangle on the screen to a bit-by-bit boolean function of three variables: its original contents (DST), a source rectangle (SRC), and a repeating bit pattern (PAT). The SUN graphics system allows all 256 possible Raster-OP functions, although only a few are used in practice.

For example, to clear the entire screen, the constant function `0` is applied to the viewable rectangle. To flash a certain window, the function `NOT Dst` is performed on that window. To write a character, the `Src` function is used, while `NOT Src` writes the character inverted (black on white), `Dst OR Src` overwrites (paints) the character, and `Src OR Pat` writes the character with a background pattern. There should be a standard graphics package to provide access at the RasterOp level.

4.2. Detailed Operation of the Graphics Board

The graphics board decodes 20 bits on the Multibus memory address lines, in the fields shown in Figure 4-4. By encoding these operation bits in the address, repetitive operations like generalized rasterOps can be done very quickly. There is a patent pending on this design.

Up to eight graphics boards may share a single Multibus backplane, with the high 3 bits selecting the board. Each board occupies 128K bytes of Multibus memory space.

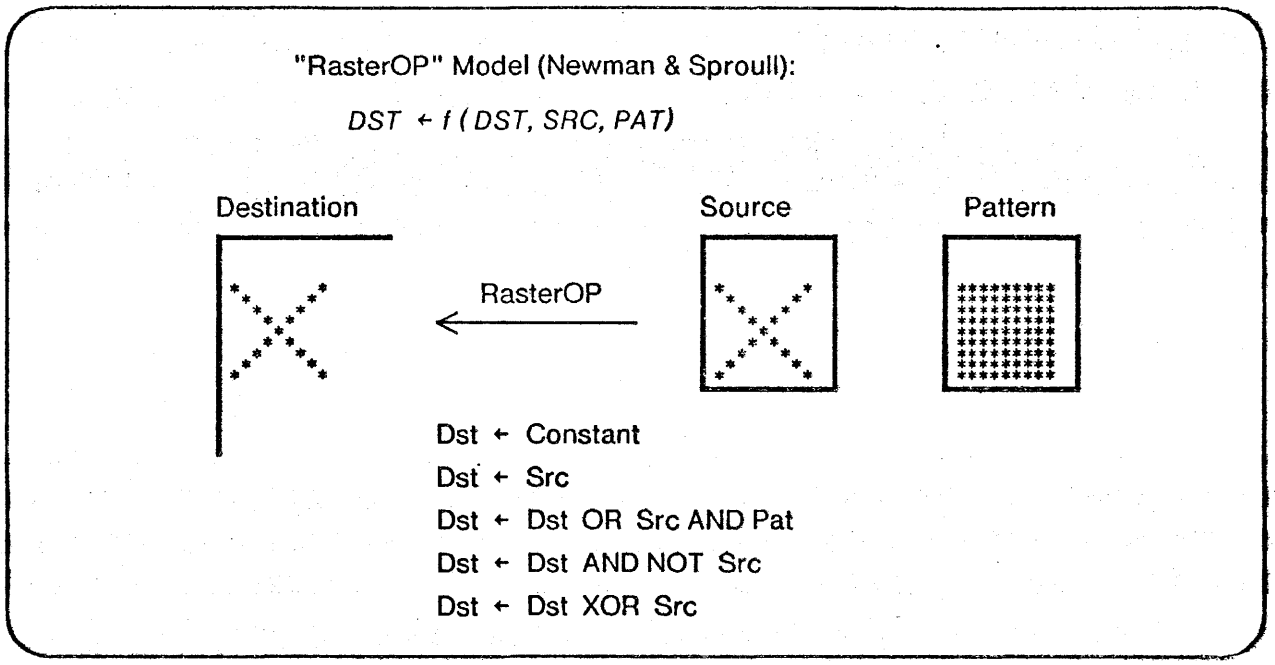

Figure 4-3: "RasterOp" Concept

Some mnemonic definitions for the frame buffer can be found in the `framebuf.h` header file (see section 2.3 for more information). All the symbols begin with the letters `GX`. To perform an operation on the graphics board, you must combine the bits together, cast the result to a `short*`, and reference the pointer. There are also a few combined symbols that can appear in C assignment conexts. See the end of this section for some examples.

The `GXupdate` bit (bit 16) is on if the frame buffer is to be modified. Usually several operations are perfomed with this bit off, to set up the control registers and one of the coordinates. Then this bit is set to actually perform the desired modification of the frame buffer.

Bits 14 and 15 select the operation. If they are set to `GXnone` then the data on the data bus is not used (although an X or Y address may be loaded in this cycle). If they are set to `GXothers` then one of the four control registers will be written with the data. If they are set to `GXpat`, the pattern register (sometimes called the "mask") will be loaded from the data bus. If they are set to `GXsource`, the data bus is loaded into the "source" register. This is the normal case for copy operations.

When `GXothers` is specified, the control register number is given in bits 1 and 2. `GXfunction` loads the function register from the low-order eight bits of the data bus. The function register can be thought of as a bit

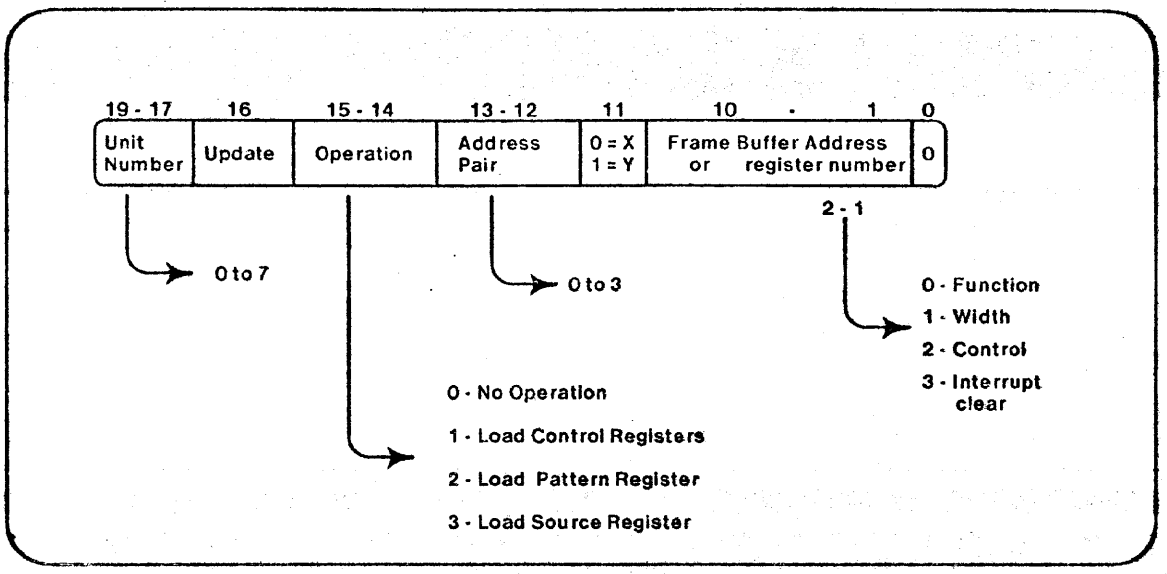

Figure 4-4: Graphics Board Address Decoding

vector, indexed by 2^0Dst+2^1Src+2^2Pat. For example, `GXinvert`=0x55 inverts the destination, `GXcopy`=0xCC copies the source to the desitination, and `GXclear`=0 clears the destination.

`GXwidth` is the register that determines the width of the RasterOps. It is loaded from the low order 4 bits of the data bus, with 0 meaning 16, so its valid range is from 1 through 16. If it is less than 16, the high-order bits of the data in the source and pattern registers will be significant on RasterOps. `GXcontrol` loads the interrupt level (low order 3 bits), interrupt enable, (bit 3, symbol `GXintEnable`), and video enable (bit 7, symbol `GXvideoEnable`) bits from the data bus. Accessing the last control register, `GXintClear`, must be done once after every video refresh interrupt to clear it, when it is enabled.

There are four pairs of ten-bit address registers (sometimes called "cursors"), selected by bits 12 and 13. Bit 11 selects either X or Y of the pair, and bits 1 through 10 of the address are loaded into the selected address register. Note that every read or write reference to the graphics board has to load one of these address registers, while it might or might not (depending on the `GXupdate` and operation code bits) modify the frame buffer.

The low order bit (bit 0) of the address must always be zero. This design was meant to be used efficiently with auto-incrementing addressing modes. For example, the following function displays an 8 by 8 "cursor" at the given position:

```
DisplayCursor( x, y )
  short x, y;           /* screen coordinates of upper left corner */
   {
      static short cursor[] = { 0x9200,
                                0x5400,  /* Left justified bit array */
                                0x3800,
                                0xFE00,
                                0x3800,
                                0x5400,
                                0x9200,
                                0x0000};

     register short
                *cursorPointer = cursor;
                *xPointer = (short *)(GXUnit0Base | GXselectX),
                *yPointer = (short *)(GXUnit0Base | GXupdate
                                      | GXsource | GXselectY);

     register short junk;                       /* something to move */

     GXwidth    = 8;
     GXfunction = GXinvert;

     xPointer += x;
     yPointer += y;

     *xPointer = junk;

     *yPointer++ = *cursor++;
     *yPointer++ = *cursor++;
     *yPointer++ = *cursor++;
     *yPointer++ = *cursor++;          /* Each of these is one */
     *yPointer++ = *cursor++;          /* 68000 instruction */
     *yPointer++ = *cursor++;
     *yPointer++ = *cursor++;
     *yPointer++ = *cursor++;

   }
```

5. The Motorola 68000 Design Module

The MC68000 Design Module is a board with an MC68000 processor, memory, and some minimal I/O devices, for designing and evaluating the MC68000 processor. It was deisgned for the EXORciser development system, but we use it with a serial connection to a VAX and a standard terminal.

Before actually using the 68000 design module, read the Design Module User's Guide [12], a black booklet by Motorola that describes MACSbug, the monitor that resides in PROM on the design module, and the I/O devices available.

5.1. Preparation of Programs

Write the program as an ordinary C program, called *<name>*.c. Do not rely on <stdio.h> - it does not apply to the Design Module. [Vaughan has rigged up some of the C library for the design module, and it should be referred to here.]

Section 2.3 describes some header files that can be included in C programs, and section 2.4 describes some libraries available. As described in section 2.2.1, you probably want the -vm option on the cc68 command line.

5.2. Compilation

To compile a single program which can be loaded into the 68000 design module (the Motorola-supplied board), use the command

```
cc68 -vm -d -o <name>.dl <name>.c
```

Error messages will be printed on your terminal. Errors may arise at any of several stages: preprocessing, compilation, assembly, and loading. To understand these stages see section 2.2.1; for the moment assume that you have got your program to compile successfully on the VAX. You now want to down-line load and test your program, which you will find in your working directory as *<name>*.dl.

5.3. Down-line Loading

Downloading is the process of getting your program into the 68000 from the VAX or wherever you keep your program. While it is necessary to compile on the VAX, the result of compilation can be put on any computer. Get your 68000 to the point where you can talk through it (in transparent or P2 mode) to the computer where your file is. Type ↑A to return to MACSbug (the 68000 board monitor). Note that this sends ↑A↑X to the host; before downloading you will need to compensate somehow for this. A carriage return will suffice on most systems. To send a return to the host without reentering transparent mode, type

(asterisk return) to MACSbug. This will send the return. In general typing

```
*text
```

will send the text to the host; you will not see the reply if any. See section 5.8 for more information.

The downloading command is RE (for REad). This command waits for the host to start typing out your file; when this happens it reads it into memory, then at the end halts and types *. Normally REad checksums each line of input; each failing line generates an error report on your terminal.

RE may take any of the following arguments:

=*text* As it takes a certain dexterity to persuade the host to delay typeout until you do the ↑A*<return>RE there is an option to send a line of text to the host as part of your giving the REad command. Thus if you say to Unix

```
RE =dl1 foo.dl
```

then the file `foo.dl` will be loaded into the 68000. The `dl1` program simply pads each line with a few nulls to allow loading at 9600 Baud.

X This option displays the data being read. It permits you to watch the file being loaded so that you can see what progress is being made. For example if the host dies (so becoming a heavenly host) you will have no indication of this without the X option. A disadvantage of the X option is that any checksum error report will disappear off the screen within 24 lines of typeout, so that you must watch the whole loading process if you want to be sure of catching checksum errors.

-C This option ignores checksums. Useful only on rare occasions.

With all options, your command would look something like

```
RE -CX=dl1 foo.dl
```

5.4. Running

To run the program type

```
G 1000
```

to MACSbug. If the program terminates normally it will type an asterisk, the MACSbug prompt, to indicate that it has returned to MACSbug.

If G is given without an argument, execution starts with the virtual PC (contents of location 400, see below). This permits an interrupted program to be restarted from where it was interrupted.

5.5. Debugging Aids

MACSbug offers debugging facilities that are moderately well documented in the design module manual. Here are the highlights.

5.5.1. Display

You may display the contents of any register merely by typing its name. Names are `PC SR SS US D0 D1 ... D7 A0 A1 ... A7`. You may see all D registers by typing `D`, and similarly for A.

To Display Memory, type:

```
DM <address>
```

(in hex, all values are in hex). The 16 bytes starting with that location are typed out. To see more than 16, supply the number as a second argument. Thus `DM 1200 100` will fill most of your screen with bytes 1200 to 12FF.

The memory values you display are real, but the register values are virtual. The register values are those that held when your program was last interrupted; they are kept in memory locations 400 to 44B inclusive, in the order PC SR D A US (where A7 is taken to be SS, the system stack pointer, rather than US, presumably since it happens to be dumped while in system mode).

5.5.2. Setting

To set register *R* to value *V*, type *R V*, as in `D3 247`. These settings will take effect on the processor proper as soon as you type `G`. To set all of the D registers, say `D:` and MACSbug will show you each in turn; for each you should either type return if you want it unchanged, or a value, return, if you wan to change it to that value.

To Set Memory, say

```
SM <address> <value> <value> ....
```

The values will be stored starting in that address, immediately. Values may be from 1 to 8 hex digits. Each value is stored in the next few bytes, as few as possible consistent with the number of digits (no zero suppression). Thus

```
SM 4000 12 34567
DM 4000
004000  12 03 45 67 ...
SM 4000 00004321 6547
DM 4000
004000  00 00 43 21 65 47 ...
```

5.5.3. Breakpoints

To set a breakpoint use the command

```
BR <address>
```

Your program will stop when it reaches that address. This is implemented by placing a 434F (TRAP 15) instruction at that address, not at the BReak command but at the G command. The instruction is removed after it has had its effect; it will be restored again at the next G. Note that CALL, described below, does not install the breakpoints. The effect is that you cannot see a breakpoint merely by looking at memory; to tell what breakpoints are set type BR without an argument.

To clear a breakpoint, type:

```
BR -<address>
```

To clear all breakpoints type

```
BR CLEAR
```

To see what breakpoints are set type

```
BR
```

If you should reset the computer while your program is running, all the breakpoints will stay put since MACSbug will have forgotten where it put them. When a TRAP 15 is encountered for which MACSbug has no record, it types ERROR and halts your program. You will have to fix it yourself, either with SM or RE.

You may delay a breakpoint so that it takes effect only on the nth time it is encountered by saying

```
BR <address>:n
```

On the *n*th encounter it breaks. When the program is restarted *n* is forgotten about, i.e. *n* reverts to 1, the default. To restore *n* give the command again.

To set a temporary breakpoint, one that clears itself when encountered, give the command

```
G TILL <address>
```

The address gives the stop (unless some other breakpoint or the end of the program is encountered first). The start is the virtual PC.

5.5.4. Tracing

To single step through a program, say TR. The program will begin execution from virtual PC, and halt after one instruction, typing out the trace information (see below) followed by

```
:*
```

When the prompt is :* it means that you can type carriage return as a synonym for TR, so that you can conveniently trace a series of step. If you type MACSbug commands at any point, the prompt reverts to *.

TR *n* will trace *n* steps, printing trace information on each step. To trace *n* steps without any printout at

all, do TD CL (see below).

5.5.5. Trace Display

Whenever MACSbug interrupts your program by reaching a breakpoint or after one instruction in the case of tracing, or when you type TD, it types out selected information. The default is that it types out the contents of all the registers. You may modify this default as follows. Each TD command is sticky, i.e. its effect is felt till you give another TD command.

TD CL	Displays nothing. Cannot be combined as in TD CL PC
TD ALL	Restores the default.
TD D0	Displays PC (similarly TD PC, TD SR, ... but not TD D or TD A)
TD D0.0	Do not display D0 (similarly TD PC.0, TD SR.0, ...)
TD D0.1	Display least significant byte of PC
TD D0.2	Display lower word of D0
TD D0.3	Display lower three bytes of D0 (only effective after TD CL)

The short forms are helpful in two ways: they give a less cluttered display, and they permit tracing to happen faster. TD CL permits multistep tracing (as in TR *n*) to proceed independently of the terminal speed, though not at full machine speed unfortunately. To see the final register values when the tracing stops do TD ALL then TD.

You may also turn words of memory into pseudoregisters which can then be displayed along with the real registers. To define say the 2-byte word at address 4564 as pseudoregister M5, say

```
W5.2 4564
```

You may now refer to M5 in the TD command, as in TD M5, or TD M5.1, or TD M5.0, each of which displays the appropriate amount of this 2-byte register. Pseudoregisters may not be larger than 4 bytes.

A pseudoregister may be relative to an address register, thus:

```
W3.4 4(A6)
```

The location is 4 past what A6 points to. A6 is the stack-frame base register for C programs, and 4(A6) is where the return address lives, so that you can monitor the calling address while tracing, as a supplement to the PC. Similarly the arguments are (in order) 8(A6), C(A6), 10(A6), ..., while the locals are (in order) -6(A6), -A(A6), -E(A6), ... provided they are all integers (4 bytes); compensate accordingly if not. Function arguments are always converted to the C type `int`. If the routine uses register variables, subtract 4 for each

register variable in computing these offsets.

5.5.6. Symbols

To assign the symbolic name FOO to a value, say

```
SY FOO <value>
```

To see the value of FOO say

```
SY FOO
```

To see all symbols, say

```
SY
```

See section for a discussion of the symbol table produced by the compilation/loading process. This table may be operated on with the above symbol manipulating commands. But Macsyma it is not.

5.5.7. Numeric conversions

The CV (ConVert) command will display both hex and decimal values of its argument. Preceding the argument with & or nothing denotes decimal, with $ hex. The printout observes these two conventions. Thus:

```
CV 45
$2D = &45
```

You may evaluate a sum of two numbers by using , as infix plus, as in

```
CV 45,3
$30 = &48
```

This does not work for differences or for three elements at once (do it in stages).

5.6. Symbol Tables

Symbols are generated and loaded automatically along with the program. They reside in the region from 6BA to FFF. To print out the symbol table type SY (see the section earlier on dealing with symbols from MACSbug). Loading a new program destroys the old symbols (that's the best we can do with MACSbug, sorry). Sometime an option will be installed permitting you to produce .dl files without symbols, to prevent this problem.

Resetting MACSbug also appears to make the symbol table go away. This is much less destructive than reading in another symbol table however. All that needs to be done to restore the invisible symbol table is to set the word at 576 (remember this as 24 squared, even though 576 is really hex) to the first empty word of the table, which can be found by printing out the table itself. The starting address of the table will be found in 576 (and in 572) since this is the pointer to the first free table entry; when MACSbug thinks the table is empty this pointer coincides with that pointing to the beginning of the table (572, or really 570, these are 32 bit pointers but it is convenient to work with them as 16-bit ones).

The format of each entry in the table is an 8-byte name followed by a 4-byte value. The name is left justified, padded with blanks, starts with an upper-case letter or period, and may only contain letters, digits, period, and the "$" symbol (at least if you want to be able to refer to them when talking to MACSbug). Knowing this it is generally not too difficult to spot the end of the table by relying on the Ascii part of the DM printout.

You should set 576 to the exact first free entry if you expect to be adding symbols manually, as in `SY FOO 33`. Otherwise you can err on the low side by up to 11 bytes when in doubt or haste.

5.7. Disassembly

The `DM` command is not much use in following code in memory. To make it easier there is a disassembly program, `dasm.dl` on `/usr/sun`, which when loaded will run in the 6000-7000 region of memory. Provided your programs do not reach into that region you can leave the disassembler there permanently.

To inspect locations starting from *n*, enter *n* into D0 and type `G 6000`. You can now single-step through the code by typing any character but `Q`. Each step will display the next instruction and the address it is at. The program will return to MACSbug either on encountering an illegal instruction or on your typing `Q` (for Quit - it must be capitalized, but that's how you have to talk to MACSbug).

Dasm understands symbols if present in the symbol table. When loaded it brings its own symbols with it, so you have to load the program to be debugged after you've loaded Dasm. (This will be fixed soon. [Ha! - Ed.])

5.8. P2/↑A

P2 mode on the 68000 is a software and hardware combination that splices out the 68000 from the terminal-host path it normally intercepts. This is accomplished when you type P2 to MACSbug by setting the RTS bit of ACIA1 high. Random logic attached to this bit then routes input to one serial connector of the board directly to the output of the other serial connector. Terminal-host traffic then proceeds at a rate dependent only on the host and the terminal. MACSbug monitors the terminal to host traffic (it is physically possible to read the traffic in either direction) and exits from P2 mode when `↑A` is seen. This assumes that the speed of ACIA1 is set to agree with that of the traffic, which it need not be. The `↑A` is the default escape; to set some other escape, say `↑V`, type

```
P2 ↑V
```

Since the `↑A` is seen by the host and MACSbug simultaneously, there is no way in this arrangement to prevent sending the escape to the host, so like any wise jailbird you should plan your escape carefully. MACSbug adds

insult to injury however by transmitting ↑X to the host after seeing the ↑A. Unlike the ↑A, the ↑X is not a default but is in Prom, so learn to live with it.

If you run your terminal at 9600 baud and the host at 1200 baud, a sensible arrangement for people at home, you will hit the obvious snag when you use P2. As should be clear from the above discussion, while you must change your terminal's speed, you need not change ACIA1's speed.

A more "brute force" way of getting out of transparent mode is to press the reset button on the design module (the black one). This will clobber the registers and reset MACSbug, however. Pressing the red abort button does not leave P2 mode, but makes MACSbug forget that P2 mode is on, leaving MACSbug in a confused state.

5.9. Memory Layout

Memory in the Motorola Design Module is laid out as follows:

0-3FF	Interrupt vectors
400-6B9	MACSbug variables
6BA-FFF	MACSbug symbol table
1000-7C00	User space
7E00-7EFF	User stack (more may be taken if necessary, clearly)
7F00-7FFF	System stack (the system/user distinction is vague on this board)

6. An Insider's Guide to SUNet

This chapter describes the current state of network communication around here. Ethernet is used to connect computers within the same building, with repeaters and gateways being used to temporarily extend the Ethernet between buildings. Arpanet connects several machines, and some are connected to Telenet, Tymnet, and other ad-hoc networks. Eventually we hope to provide other communications media for the intermediate distance needs, like 10 Mb standard Ethernet, Fiber-optics, point to point serial links, packet radio, or a broad-band cable TV compatible system. The rough topology of the network is illustrated in Figure 6-1. For more information see the references [7] and [16].

This information is really quite volatile, so it is imperative that this chapter be kept up to date. Each description below consists of the computer you want to come from, followed by a colon, and the command that invokes the given program. A short description includes the hosts you can go to with that program.

6.1. Remote Terminal Programs

VAV: `telnet` *host*

Pup user telnet [5] to any Pup host with a telnet server running. Currently this means Shasta, Diablo, Helens, Lassen, DSN, and Sail. When the gateway is working you can get to Sumex and "Tiny," the 2020 at the Medical Center. Documentation is obtained with the `man telnet` command. `Telnet` has transcript and shell escape options. Type code 036 followed by c to get out (usually control-↑ or control-shift-N). Pup telnet has gateway capabilities to the Arpanet. If the host name you give is not in the PUP network directory, or there is no PUP route known to it, it will connect to the telnet gateway at Sail and try the Arpanet.

Sail: `r chat` Pup user telnet as above. Control-meta-q to get out. Currently it simulates a very simple terminal, so you can't even do things like backspace.

Alto: `telnet` *host*

Pup user telnet also, based on the chat program. Simulates a simple screen terminal, on which vi almost works. Also available as a boot file.

Alto: `talk` *host* The `talk` program gives you multiple windows (stacked vertically), and you can run emacs or vi in them, so it is currently the default for Unix telnet servers. You can create a window of 52 lines, and then specify `term bigtalk` to Unix to get a big screen.

Alto: `dmchat` *host*

Another terminal program using Pup telnet protocol, based on the Chat program. This one gives you one big "datamedia" window, which works well with Sail. If you are using `dmchat`, tell Unix you are a `dm2599`. You must have the fixed width font `snail10.al` on your disk for this to work (or an entry on your `user.cm`).

Sail: `dial diablo`

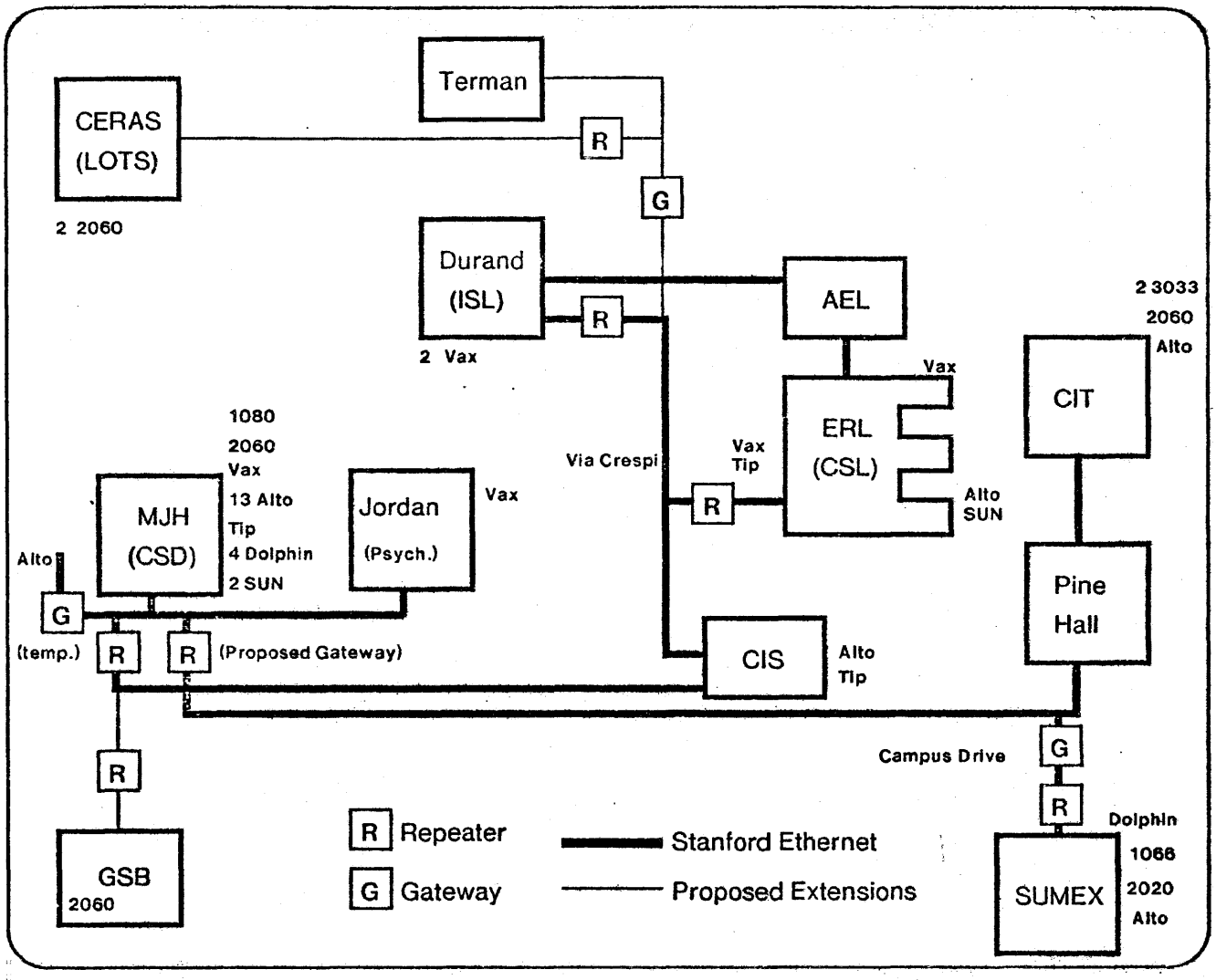

Figure 6-1: Topology of SUNet

Simple 1200 baud serial line to Diablo. Loses characters routinely. Use only when all else fails (which is all too often these days). Get out with control-meta-q.

VAX: `sail/score`

sail runs on Diablo, score runs on Shasta. 1200 baud serial link. Use `telnet sail` or `telnet score` instead.

Score: `telnet shasta`

This currently goes through Sail's telnet gateway. Typing this command will be guaranteed to get you there in the best possible way (directly over the ethernet in the future). There is also a possibility of using the 1200 baud serial link. This and the score program are mutually exclusive. Loads down score badly. Exit with control ↑ c.

6.2. File Transfer Programs

VAX: `ftp` *host* Pup user ftp. Talks to any Pup server FTP, currently this means Shasta, Diablo, Helens, Lassen, DSN, Sumex, Tiny, and any alto running ftp. Documented with `man ftp`.

Sail: `r pupftp` Pup user ftp. Same as above. Experimental. Preferred way of getting files from VAXen to the outside world. Use `type X` for 32 bit binary files. Documentation in `pupftp.tvr[up,doc]`.

Alto: `ftp` *host* Another Pup user ftp, as above. Prefered way of getting files to Altos.

Alto: `eftp` Simple transfer of files from Sail. Self-explanatory. Now mostly outdated by ftp. It is still used for sendind files to the Dover, but that should be done invisibly to the user.

Sail: `r eftp` Old, slow, but reliable way to get binary files between sail and ethernet hosts. Say `i` for mode, using `1` for left hand bits.

VAX: `eftp e[xr]` *host filename*
Obsolete way of getting binary files to/from sail. Be sure to do a man eftp command, since there are many obscure options. We ususally use `e` and often `s` for binary files and `ct` for text files. For text files the command `vtos x` (for VAX-to-Sail) abbreviates `eftp reqtlv sail x`, while `stov` (for Sail-to-VAX) abbreviates `eftp xeqtlv sail x`.

Sail: `ftp` *Arpanet-host*
Arpanet ftp. Remember to use ascii mode or right curly braces (as well as other random characters) will disappear.

Score: `ftp` *host* Prefered way of shipping files around the Arpanet.

6.3. Walk Net (Tape transfer)

VAX: `ansi [load] [init vol] [write` *files*`]`
Reads ANSI standard labeled tapes. Works with VMS, RSX, RT-11, `tapeio` (with `/Format:Ansi`), and `vaxtap` tapes. Text files only. Documented with `man ansi`

Score: `vaxtap` Writes and reads ANSI standard labeled tapes for VMS systems, also works with the *ansi* program on Unix. Unfortunately it does not handle wildcards, so you need .MIC files for large transfers.

Score: `tapeio` Writes and reads ANSI standard labeled tapes for CIT, also is supposed to write DEC compatible labeled tapes. You can exchange tapes with the `ansi` program with the `/format:ansi` option. You must run the `tapelabel` program on a new tape to write a label.

Score: `<su-net>taput`
A really kludgy way of getting binary tapes to VAXen. Use ftp to sail and `pupftp` instead.

7. SUNOS - A Small Operating System

SUNOS is a compact operating system for the Sun workstation. It manages asynchronous processes, the memory map, and interprocess communication.

Process Services

`spawn( function, argcount, arg1, arg2, ...)`
spawn new process, with stream args

`wait(p)`
wait till p holds (non-busy waiting)

`terminate()`
terminate this process

Stream Services

`char * getc(r) char *r`
get char from stream *r*

`getcto(r, v) char *r,v`
get from stream *r* into char *v*

`putc(c, w) char c,*w`
put char *c* on stream *w*

`int emptyc(r) char *r`
predicate: stream *r* is currently empty

`char *cstream()`
create a char stream and return its writer

Corresponding stream services exist for other types of streams, namely short, integer, and reference, for which the respective abbreviations `s`, `i`, and `r` are used in place of `c`(character).

Storage Services

`char * create(n)`
create reference to start of new *n*-byte buffer

`char * dupref(p) char *p`
duplicate reference *p*

`dispose(p) char *p`
dispose of reference *p*

The put and get primitives use ordinary `char*` (reference) variables. Thus, to permit two processes talk to

each other, use `cstream` twice to produce two references, one pointing into each of two new streams, and then apply `dupref` to each reference to produce two more references for reading from those streams.

Processes may pass references around freely, provided they observe the discipline of using `dupref(`*p*`)` to make duplicate copies of references and `dispose(`*p*`)` to dispose of those duplicates. A reference may be used either for reading or writing so long as it is consistently used for that. Only one reference to a given stream may be used for writing, but any number of independent references may be used for reading, permitting independent processes to read the same stream each at its own speed.

A file using SUNOS should `#include "sunio.h"` and should contain a sunos() function, which should be confined to spawning an initial set of processes and creating and passing streams to them. The spawned functions are animated (brought to life) after sunos() exits. There should be no `main()` function in the file, this being provided by the library. If the user does provide a `main()` it will be substituted for the library-supplied main, in which case to ensure an orderly startup of sunos it should be modelled on the library version, for which the source is in `/usr/sun/sunos/sunos.c`.

The library supplied `main()` initializes the stream and process managers and the interrupt handlers, then calls `←strtcal` to start up calloc, and finally exits to calloc by executing a sleep. `←strtcal` in turn initializes calloc and calls the user-supplied sunos() to create the initial processes, which come to life after `main()` goes to sleep.

The char streams stdin and stdout are predefined and available to user programs. They are initially connected to the terminal I/O of the Design Module. They may be set to other streams as desired; their old values should be saved somewhere if it is desired to continue to communicate with the terminal. The functions putchar and getchar are defined in terms of putc, getc, stdin, and stdout, as usual.

An example SUNOS program may be found in /usr/sun/sunos/example.c. This example implements a quizmaster and a quizkid who fire streams of questions and answers at each other.

The cdm command will produce the corresponding .dl file, which includes the code from pman.c, strm.c, and calloc.c. This file may be loaded and started in the normal way, namely via G MAINE. During the settling in period, the SUNOS symbols will be loaded along with the user symbols to help the user distinguish his bugs from SUNOS's, but eventually they will be omitted in the interests of avoiding cluttering up the user's table.

SUNOS is designed to perform best under heavy load. The total overhead to send and receive one datum

on a heavily used stream, averaging in the overhead of context switching and buffer switching, and assuming use of register variables and the readcharto (as opposed to readchar) primitive, is roughly 20 microseconds on an 8 MHz 68000.

The present versions of strm.c and calloc.c are close to final. However the present version of pman.c is too trivial to be really robust; the calloc.c facilities, as described in section 7.5, support much more robust process managers.

Sunos implements processes (virtual concurrency), stream-based interprocess communication, and storage management for the Motorola 68000 computer. Although the design was done specifically for the 68000 to avoid all compromises that portability considerations might have entailed, the semantics of the resulting product turned out to depend only on a few architectural features common to many computers, namely the existence of program counter, stack pointer, and status register, and the use of interrupts to schedule "external" processes.

This overview considers Sunos from a perspective midway between that of a user and an implementor. (Implementors sometimes have difficulty presenting a pure user's view too soon after completing the implementation.)

Sunos provides the following services.

7.1. Process-oriented Services

Processes provide virtual concurrency. A process may be defined to be a stack (including its current contents) together with the current state of the processor.

Processes fall into two categories, internal and external, distinguished by how they are scheduled by Calloc, the Cpu ALLOCator.

7.1.1. External Processes

External processes are scheduled by interrupts and are not even known about by Calloc. They run in system state, preempting the CPU when their interrupt occurs. Each external process is responsible for restoring the CPU to its original state on exit. External processes communicate with internal processes via shared memory. The stream services are available to them, but they may also use more block-oriented forms of communication.

7.1.2. Internal Processes

Internal processes share the processor under the control of Calloc, a round-robin preemptive scheduler. An internal process may voluntarily surrender the Cpu at any time; otherwise at the expiration of its time quantum the Cpu will be preempted.

Internal processes have two components, whimsically called the patient and the physician. The patient is the normal part of the process while the physician acts as exception handler. The physician permits diagnosis and debugging of erring processes, and is also responsible for preemption, permitting individual implementations of both blocking and nonblocking mutual exclusion methods that the process may need to survive with Calloc's asychronous preemptive scheduling. (This does not needed for Sunos's stream facilities, which incorporate their own nonblocking mutual exclusion.)

7.1.3. Patient services

spawn(p,n,s1,...,sn)
: spawn process p with n streams s1,...,sn

wait(p)
: wait until p holds (non-busy waiting)

terminate()
: terminate this process

error(e)
: commit error e

Medium-level atomicity is also provided for. The patient may at its option run atomically for a limited time. In this state preemptive scheduling by Calloc, though not by interrupt driven processes, is inhibited. The patient enters atomic state by signalling its physician, typically by setting a global variable called Lock, and leaves atomic state by another signal to the physician, typically by clearing Lock.

(Medium-level atomicity provides an attractive alternative to locking resources in use by a preempted process. Such resource locking is both expensive and a source of deadlocks. At the moment of preemption it may often be cheaper to finish the critical section than to incur the costs of blocking a process sharing the resource presently accessed by the preempted process. Higher levels of atomicity may be provided in the usual way, with the usual deadlock problems, using medium level atomicity to implement synchronization primitives, for which low-level atomic instructions such as test-and-set may not always be powerful enough.)

7.1.4. Physician services

The physician services include all patient services except error(e), together with:

`status()`
: supply physician with patient's pc,sp,sr,error code.

sstatus()
: set patient's pc,sp,sr (rest is directly accessible)

run()
: run patient normally

singlestep()
: run patient for one step

breakpoint(a)
: set breakpoint at instruction address a

The status and sstatus services are needed in case the physician has no other way to access the patient's program counter, stack pointer, status register, and error code (reason for invoking the physician). The rest of the patient state is accessible

Physicians are expected to be implemented with system-supplied routines as a rule, and are not themselves candidates for diagnosis and debugging; this is a strict two-level approach rather than a hierarchical one.

The physician inherits all the capabilities (read-write-execute access rights) of its patient, together with write capability for the patient's state (cpu registers and stack) and read capability for all code for which the patient has execute capability. The breakpoint service provides an additional capability for code.

7.2. Stream-oriented Services

Streams provide a form of communication between processes in which one process may write a sequence of data into the stream and any number of processes may read that sequence from the stream asychronously. The data types supported by Sunos are characters (one byte), shorts (two bytes), integers (four bytes), and references (four bytes, garbage collected). Streams are type-homogeneous: only one of the four possible types of data may appear in a stream.

Streams are accessed by stream references, which are values of type either char*, short*, or int* in the C sense. Sunos provides the equivalent of *s++ = d for storing a datum d into a stream referenced by s, and *s++ for fetching a datum from a stream referenced by s. (It is not possible however to use *s++ = d and *s++ directly due to discontinuities in the internal representation of streams and the need to garbage collect

streams.)

Stream references have many of the attributes of ordinary pointers. They may be assigned to variables, passed as arguments to functions, and returned from functions. In passing them around however they may not be duplicated implicitly, but must be copied explicitly with the Sunos function dupref(s). Thus if the assignment s2 = s1 is performed and then both s1 and s2 are subsequently used as references to the same stream, the assignment must be rewritten as s2 = dupref(s1). Pointer arithmetic is not permitted, nor may pointers be compared with each other since distinct pointers may point to the same place in a stream.

Streams are defined entirely independently of Calloc. They do not take advantage of the atomicity catered for by Calloc but rely on nonblocking methods for accomplishing mutual exclusion at all potential interaction sites. This permits streams to be used in common by both internal and external processes, reducing the variety of communication primitives required in the system. It is possible to perform all interprocess communication via streams. It is intended that even transactions that might ordinarily be handled for the sake of efficiency by block moves be handled by streams for the sake of uniformity of communication. The efficiency question is intended to be ignored by the user and taken care of at the implementation level (see section D below).

Stream services fall into two categories, transactional or data-oriented, and existential or identity-oriented.

In the following, T denotes the type of the stream, and may be any of c, s, i, or p, each abbreviating one of **the four types.**

Transactional Services

putT(d,s) — put datum d in stream s. Stream version of *s++ = d; getT(s)
get datum from stream s. Stream version of *s++ emptyT(s)
return 1 if stream s is empty, else 0

Existential Services

makestreamT() — create a stream, return a reference to its start dupref(r)
make a copy of reference r (a typeless operation) dispose(r)
dispose of reference r (a typeless operation)

Semantically dupref acts as an identity operation, apparently merely returning its argument. Behind the scenes a reference count is incremented. When dispose is called a reference count is decremented. Reference counts permit garbage collection of storage associated with streams, permitting the user to ignore the details of storage management for streams.

7.3. Performance Services

Performance services are services that are semantically redundant, i.e. are already supplied by other services, but that offer alternative tradeoffs in time and space. The advantage of having performance services is finer programmer control of program performance. Their disadvantage is that program structure may be obscured by the clutter of performance details interwoven with those aspects of the program that contribute to its semantic correctness.

The performance services are as follows:

`block()`	temporarily suspend execution of this process
`sleep()`	destructive block: d0-d7,a0-a6 not preserved on return
`putTto(`*d*`,`*s*`)`	same as `putT(`*d*`,`*s*`)` if evaluation of d has no side effects
`getTto(`*v*`,`*s*`)`	equivalent to `v = getT(`*s*`)`

`block()` is semantically equivalent to the empty statement. Its effect is to temporarily surrender the processor to another process. Eventually the scheduler will return the processor unharmed to the blocked process. `block()` is used by wait(p), which is implemented as `{while (!p) block();}`.

`sleep()` is a variant of `block()` which does not preserve registers d0-d7 or a0-a6. On the Motorola 68000 `sleep()` is implemented as a system call, `trap #2`, while block is implemented as push-sleep-pop, namely

```
movem1 #/FFFE,sp@-
trap #2
movem1 #/7FFF,sp@+.
```

The two `movem1`'s together require 290 68000 cycles, or 36 microseconds on an 8 MHz 68000. Thus context-switching efficiency may be improved substantially by use of `sleep()` together with alternative ways to preserve needed registers such as saving them most of them on entry to a body of code that executes `sleep()` repeatedly. For many applications however the load-buffering capability of Sunos should make context-switching costs negligible under heavy load. Hence `sleep()` should be reserved for situations involving short back-and-forth exchanges between processes; `block()` will be adequate for more heavily stream-oriented tasks.

`putTm(`*d*`,`*s*`)` is a variant of `putT(`*d*`,`*s*`)` which is expanded in-line instead of as a subroutine call. Since d is referred to more than once in the expansion, evaluation of d should not have side effects. For example putcm(getc(s),t) will not be equivalent to `putc(`*d*`,`*s*`)` because `getc(`*s*`)` has the side effect of assigning to *s*.

7.4. Performance Characteristics of Present 68000 Implementation

The performance figures of greatest interest are:

- Cost of nonpreemptive (process-initiated) blocking
- Cost of preemptive scheduler-initiated blocking
- Cost of preemptive interrupt-initiated blocking
- Cost of a stream-mediated transaction (combined put and get)
- Frequency of each of the above.

Nonpreemptive blocking using `block()` requires 34 instructions or about 100 microseconds if we assume 2 microseconds per instruction. The non-register-saving `sleep()` instruction avoids the two moveml's that alone account for 36 microseconds; thus `sleep()` costs approximately 60 microseconds.

Frequency of nonpreemptive blocking is intended to decrease as system load increases, at least for tasks that lend themselves to batching. The use of streams in interprocess communication will often ensure the automatic batching of tasks; a process will continue to process data coming from a stream for as long as data is available, blocking only when the stream becomes empty.

Preemptive blocking is performed by requesting the physician to "retire" this process. Normally this costs 40 instructions more than blocking, i.e. 74 instructions or about 180 microseconds. If the process was running atomically then the overhead is increased to that required to single step the process through the atomic section, approximately 40 instructions or 80 microseconds per step.

Frequency of preemptive blocking is intended to be very low in comparison to the overhead of preemptive blocking. This motivates the choice of a 10 millisecond quantum for Calloc, making the 180 microseconds required for preemptive blocking negligible. Whether the additional overhead of leaving an atomic section increases this substantially on the average depends on the probability of being in an atomic section at the time of preemption. In general this can be assumed to be less than 0.1, and atomic sections can be assumed to be less than 10 instructions long. Then the expected overhead attributable to atomicity at preemption is at most 0.1*10*0.5*40 = 20 instructions or 40 microseconds (the 0.5 assumes uniform distribution of where in the atomic section preemption was requested).

Interrupt-mediated preemption is extremely cheap: essentially the cost of the exception. The external process saves only those registers it needs immediately, which is the same protocol observed in the C calling sequence for ordinary subroutine calls, whence should not be counted as an additional overhead.

The cost of reading and writing data in a stream decomposes into typical and boundary (buffer-discontinuity) costs. We first give the costs if the efficiency oriented operations are used, assuming the stream pointers and data are held in registers. The typical cost of writing a datum is 4 instructions, while the boundary cost is 90 instructions. The typical cost of reading a datum is 2 instructions while the boundary cost is estimated at 40 instructions though has not been checked. With buffers holding say 240 data, the average cost of a write-and-read transaction is then 4 + 2 + (90+40)/240, or approximately 6.5 instructions. Measurements suggest that the cost of a transaction along with a minimal amount of computation on the data being transacted (adding up a stream of chars) requires about 20 microseconds on an 8 Mhz 68000, which is in line with the analytic prediction if we allow 7 microseconds for the associated computation.

7.5. Calloc - A CPU Allocator for the Motorola 68000

7.5.1. Overview

Calloc, for *Cpu ALLOCator*, is concerned with allocation of the CPU to "internal" processes, defined as processes running in user state at interrupt level 0. In standard operating system terminology this would be called a scheduler. Calloc and its senior sibling process, Malloc the *Map ALLOCator*, run in system state at interrupt level 0. The only other processes are the interrupt driven processes or IDP's which run in system state at positive interrupt levels, preempting the cpu according to their level. An IDP is visible only to those processes with which it shares memory, which normally will be one or more internal processes on the distal end of a stream shared with the IDP. Calloc is entirely unaware of the existence of IDP's.

Calloc has a minimal set of duties which require little code to implement and can be easily performed. As such Calloc should be of interest to operating systems theorists as a tractably small object of study, and to operating systems implementors as a small, easily implemented, efficient, and effective component of an operating system.

Calloc is dedicated to maximal autonomy of internal processes. The objective is to minimize duties performed in system state, thus minimizing the chances of global system failure due to errors committed in (vulnerable) system state. The autonomy is supported in two ways, one negative and one positive. On the negative side Calloc forgoes the luxury of access to any space but its own, thereby reducing the vulnerability of the system to system-state errors. More positively, Calloc offers enough services to permit competent user state processes to assume duties normally assigned to system state processes.

The inaccessibility of memory to Calloc turns out to present no serious obstacles to the implementation of Calloc. As it happens none of the services provided by Calloc require access to other spaces. A residual issue

of accidental map access leading to accidental memory access by Calloc remains an unsolved problem that we discuss in more detail below.

As a basis for providing services, Calloc recognizes a partitioning of every internal process into what we shall term the physician-patient pair. This recognition takes three forms: unlimited patience with "ill" (exception-causing) patients; support for physicians, limited to the kind of support best offered by a process running in system state such as Calloc; and a very contemporary intolerance for malpractice manifested as immediate and unreported termination of an erring physician along with its patient.

This scheme should work well provided only the most trusted software is used for implementing the physician half of each pair. The behavior of Calloc towards physicians and patients is the only mechanism depended on to encourage programmers to observe this discipline in writing process software. Without this discipline debugging should prove most difficult; conversely, with it the ease and efficiency of debugging should advance in step with advances in "medical technology."

To appreciate this design decision better it helps to think of physician software as requiring a level of dependability only a little less than that of Calloc itself. The reason physicians are less critical is that the penalty for physician failure, while seemingly very severe, is nevertheless less severe than the penalty for Calloc failure. A failed physician only leads to the unreported loss of a process, whereas Calloc failure can cause the unreported loss of the whole system. This also means that an inappropriate choice of physician will inconvenience only the chooser and not processes running in other spaces.

There are of course alternative methods of debugging physicians that are not supported explicitly by Calloc, just as there are methods for debugging Calloc itself. For the most part physicians themselves can be debugged by running them as patients. When running as physicians, an alternative mode of debugging is to put the physician in communication with another physician who can at least report the termination of its partner.

A different approach to the physician-patient relation is to make it hierarchical, putting physician debugging on the same level as patient debugging. The cost is added complexity in Calloc in coping with such a hierarchy. Our preference here has been to sacrifice a certain amount of convenience in physician debugging in favor of keeping Calloc simple.

7.5.2. Machine Dependencies in Calloc

The initial Calloc design is tailored to the SUN-1, a Motorola-68000-based CPU board. The only features of this board on which Calloc depends critically and which are not features of essentially every CPU board are the 68000's trace facility, without which the duties of physicians would require substantially more assistance from Calloc, compromising system robustness; and the SUN-1's per-page protection capability, on which Calloc depends to protect physicians from their patients and both from Calloc, and to make physicians invisible to their patients. To within these details, the principles of Calloc should be found to be broadly applicable.

Ideally Calloc should be the junior sibling to Malloc, in that Calloc should have no say in how memory is allocated to processes and should not be able to interfere with Malloc's duties, in contrast to Malloc's absolute freedom. Unforturnately the SUN-1 does not support this level of protection since map access and hence unrestricted memory access is granted to all system state processes, not just to Malloc as we would prefer. Thus the promised protection of internal process memory from Calloc is only weakly achieved, by not having any Calloc code that intentionally references the map. Accidental map references remain an unfortunate possibility that we would hope future hardware would make it easier to avoid. The absence of iterative constructs and address register arithmetic from the Calloc code should help to reduce the likelihood of such unintentional map accesses. One consolation is that at least Calloc is no worse in this respect than any other system-state schedulers.

7.5.3. Calloc Duties

Calloc attends to the following.

1. Requests for animation (bringing to life) of a given process, that is, giving it a share of the cpu.

2. Requests for sleep, that is, temporary surrender of the cpu.

3. Requests for termination, that is, permanent surrender of the cpu.

4. Equitable allocation of the processor; at uniform intervals the active process is put to sleep and the next in turn is awoken (round-robin scheduling).

5. Automatic saving and restoring of pc/sp/cc, the process's program counter, stack pointer, and condition codes.

6. Error recovery. An erring process is permitted to try to recover on its own. No limit is placed on the number of errors made. Calloc depends critically on the physician-patient dichotomy to avoid the major problem associated with this degree of latitude, namely vegetating, the perpetual survival of terminally ill processes.

7. Requests by a physician to run or single step its patient. The physician is at liberty to set

breakpoints in the patient, which Calloc supports in a way that makes this invisible to the patient.

7.5.4. Calloc Nonduties

Many duties normally entrusted to system-state software are not attended to by Calloc, in the interests of autonomy, flexibility, and efficiency. Such duties often include the following.

1. Saving and restoring the contents of the cpu other than pc/sp/cc. Calloc assumes that each internal process will save and restore the contents of the cpu registers that it cares about.

2. Diagnosis, repair, or reporting of sick processes. The process must diagnose and report its own problems, using its physician component, and heal or terminate itself.

3. Allocation of memory. This is handled at two levels on the SUN-1, as a consequence of details of its memory structure. At the higher level, SUN-1 memory is structured into possibly overlapping spaces of varying sizes. Accordingly there exists Calloc's aforementioned companion Malloc, which like Calloc runs in system state at interrupt level 0. At a lower level is conventional memory allocation within a space, which is handled by the user processes themselves.

4. Allocation of the processor to interrupt-driven processes (IDP's) (introduced at the start of the overview section).

5. Creation and destruction of processes. Calloc draws a Pinnochio-like distinction between creation and animation, taking responsibility only for the latter. Process creation and destruction are left to user processes. The quality of the created process is up to the author of the creating process; in general the best effects should be obtained by creating processes with highly trusted physicians. Process destruction entails primarily deallocation of resources, which though it may be handled by any component of the process, is for greatest reliability left to specialist programs running in user state.

6. Reading or writing of internal process memory. Calloc does not reference internal-process memory, making it independent of decisions about how system state processes access memory normally accessed in user state. Although the SUN-1 makes system-state access to user-state memory as easy as access to system-state memory, future boards may vary this. Calloc is immune to such variations.

7. Capability management. No global system of capability management is envisaged for SUNOS. The definition and use of capabilities is intended to be a matter of agreement between consenting processes. The set of rights and duties of internal processes from Calloc's point of view provides an example of this. This philosophy is part of the overall decentralization philosophy of SUNOS.

8. Interprocess communication. No global system of interprocess communication is envisaged for SUNOS. Instead it is expected that a variety of interprocess communication protocols will be used, one or two of which may come to dominate on account of their combination of general applicability and high efficiency.

Calloc offers enough services to permit quality care of the patient by its physician, including:

- Full access by the physician to the state of the patient.
- Protection of the physician from the patient's illness;
- Invisibility of the physician to the patient;
- A single step facility to help the physician diagnose and/or treat the patient.

Control is passed from the patient ("normal" status) to the physician ("suspended" status) when an error occurs.

Physician access to the patient is straightforward for all of the patient except its pc/sp/cc, error state, and space size. To obtain this information a system call to Calloc is provided which returns this information in five CPU registers. This call also grants the physician read-write access to the patient's memory.

Protection of the physician from the patient is implemented by allocating a portion of the space to the physician and denying write access to those pages while in normal status. A physician whose writable memory fits in one page (2K bytes) will require Calloc to access only one map entry for each change of status.

Physician invisibility is achieved by denying the patient any read access to the space the physician writes in, the motivation being for the patient not to see mysterious variations in memory. Seeing or even executing the physician's code is permissible provided the code remains fixed. The physician should not use the patient's stack in place of its own, except where it makes arrangements to completely erase all traces of its presence before returning to normal status.

7.5.5. Requests to Sleep

A process temporarily held up by lack of input or unavailability of some resource may temporarily surrender the CPU to Calloc. It does not get it back until all other processes have had a turn. This is synchronous sleep; as such it may not be necessary to save all the CPU registers.

Where the cpu changes hands frequently (say every 100 microseconds or less), the affected processes will arrange to minimize context switching overhead, e.g. by replacing "while (!condition) {pushstate; sleep; popstate;}" with "if (!condition) {pushstate; do sleep while (!condition); popstate;}" and treating functions containing "sleep" as though they declared all of d2-d7 and a2-a5 (a trivial mod to the compiler), saving any given register across the largest block containing the sleep but not containing a reference to that register.

7.5.6. Cleaning Up

The following approach is proposed for having physicians clean up. There are three granularities of storage involved: space, stack, and frame. A space is defined by the 68000 board's SID (Space ID) register; spaces are protected from each other except when they overlap as when sharing for interprocess communication and related purposes. Stacks are associated with processes; each process has two stacks, one for normal use and a small one, normally inaccessible, for the physician. Frames are as defined by the C68 compiler (see section 2.5).

Per space: all storage is reclaimed by Malloc when a space is abandoned (contains no further viable processes).

Per stack: Each space may contain any number of processes; these are freed by the process's physician when the process is terminated for any reason, as are other patient resources known to the physician such as stream pointers.

Per frame: No fixed scheme is prescribed here; rather each exception handler is expected to understand its patient's conventions concerning stack use. A C68-dependent convention will be to use two sources of information: the add-to-stack-pointer instruction produced by the C68 compiler after each call, and the link instruction at the entry to each function. Either of these uniquely determine the number of parameters, and together they provide a consistency check on each other in the event a6 or the stack has been badly damaged or a nonstandard calling sequence has been used without telling the exception handler.

A significant advantage of this approach is that it does not commit itself unbendingly to a particular choice of calling sequence. New calling sequence conventions may require rewriting of physicians, but do not affect calloc itself.

7.5.7. The Calloc Process Model

Calloc has a conception of a process appropriate to Calloc's duties. Calloc divides its notion of the state of the process into data state and control state components.

The data state components are:

- a value for the SID (Space ID) register, determining which region of the memory map the process has access to;
- pc/sp/cc values (program counter, stack pointer, condition code register) for each of the physician and the patient;

- the patient's error status.

The SID register value is determined by Malloc, who may also change it. Calloc uses it only to set the SID register for the current (running or retiring) process. The SID register maps a virtual address to a map address, which further maps the result to a physical address.

The pc/sp/cc values are used to initialize the cpu on exiting to the process. The patient error status is reported to the physician on request, along with the patient pc/sp/cc values.

The control state components are:

- activity: one of running, retiring, or sleeping;
- status: normal or suspended.

A running process is one in control of the CPU. A retiring process is one which is attempting to go to sleep. A sleeping process is one that does not have control of the CPU. A running or retiring process is called current; there is at most one current process at a time.

A normal process executes the patient when running, while a suspended process executes the physician. When retiring, a process executes the physician whether normal or suspended.

The process state is represented in the present SUN implementation with the following 32-byte struct in the C programming language.

```
typedef struct Procdesc                    /* process descriptor */
  {int *patpc;                             /* pat program counter */
   int *patsp;                             /* pat stack pointer */
   int *patsb;                             /* pat stack base */
   int *phypc;                             /* phy program counter */
   int *physp;                             /* phy stack pointer */
   int sid;                                /* space id */
   char patcc;                             /* pat condition code reg */
   char phycc;                             /* phy ditto */
   short paterr;                           /* patient error status */
   struct Procdesc *next;                  /* next process descriptor */
  } *procdesc;
```

The current process is identified with a variable Current of type procdesc. To facilitate removal of a terminated process the procdesc variable Last identifies the process that was current prior to the present one (with Last = Current if there is only one process). Last contains no information not deducible less efficiently from Current.

7.5.8. Control State Transitions

The behavior of Calloc towards the current process can be represented as a finite state automaton, the states being the control states of the process. This automaton has five inputs, namely error, terminate, sleep, run, and timeout. An error input may be an error exception, a trace exception, or a request to be considered to be in error. A terminate input is a request to be terminated. A sleep input is a request from the process to be put to sleep. A run input is a request from the physician to transfer process ownership to the patient, either for one step or indefinitely. A timeout input is a signal from the timer that this process's time is up.

The following state transition table covers the case when exactly one input has arrived. As we care only about the current process here, the initial states are normal running, suspended running, normal retiring, and suspended retiring.

INPUT: / STATE:	error	terminate	sleep	run	timeout
norm run.	susp run	terminated	norm sleep	susp run	norm ret.
susp run.	terminated	terminated	susp sleep	norm run	susp ret.
norm ret.	terminated	terminated	norm sleep	terminated	terminated
susp ret.	terminated	terminated	susp sleep	terminated	terminated

It is possible for any combination of (i) error, (ii) one of terminate, sleep, or run, and (iii) timeout to occur simultaneously. If error is present, then terminate, sleep, and run requests are ignored, though the request is stored as part of the error state of the patient. Otherwise multiple inputs are processed as though they had arrived sequentially in the order given in the table. Thus error and timeout together take a running patient first to a running physician and then to a retiring physician. For the case of sleep with timeout we add the condition that timeout of a sleeping process has no effect. (Thus a returning patient that gets its sleep request in just as the timeout arrives survives by the skin of its teeth.)

7.5.9. Calloc Services

We now detail the services provided by Calloc.

Animation — The function `animate(patpc,patsp,patsba,phypc,physp,sid)` requests Calloc to add to its list a process with the given pc/sp values for patient and physician and sb for patient, a zero cc for each, a zero patient error status, and a control state of sleeping patient.

Sleep — The function `sleep()` requests Calloc to put the requesting process to sleep. When the process is reawoken later, its cpu registers d0-d7 and a0-a6 may have changed, but the pc/sp/cc values will be preserved.

Termination — The function terminate() requests Calloc to terminate the requesting process. Calloc does not consider the issue of returning the stack and streams to free storage.

Status — The physician may use the function status() to extract from Calloc the entire state of the patient. This information is returned to the physician in registers d4 (pc), d5 (cc,error status,space size), d6 (sp), and d7 (sb). In addition the physician is granted read and write access to the patient's space. This access is revoked on resumption of normal processing. Use of status() by the patient counts as an error

Run — The physician may use the function `run()` to return the process from suspended to normal status. Calloc restores only the patient's pc/sp/cc; it is the physician's responsibility to restore the other registers. Calloc also restores the access control to this space in force at the time of animation of this process.

Single Step — The physician may use the function `singlestep()` to achieve the same effect as run() but in 68000 trace state. The ensuing trace exception is treated by Calloc as an error of type trace and the process is then suspended again.

7.6. The Edit-String Protocol

Edit-strings are a particular representation of strings designed to support a variety of protocols for operating on strings. The most general string editing operation, replacement, is supported, but certain special-purpose kinds of operations such as stream-read-and-write have particularly efficient implementations with edit-strings. All references to edit-strings are via ordinary pointers to individual data within a string.

7.6.1. The Edit-String Data Structure

Storage is divided into buffers each of size a power of two, each aligned by its size. Each buffer contains a header and a body. The header is structured into the following fields.

```
typedef struct Bufhead          /* Buffer head definition */
{struct Buffer *next;           /* next buffer */
 struct Buffer *prev;           /* previous buffer, if any */
 char size;                     /* log base 2 of size of buffer */
 char type;                     /* type of data in buffer */
 short llim;                    /* left end of data in buffer */
 short rlim;                    /* right end of data in buffer */
 short refco;                   /* number of owners of this buffer */
} *bufhead;
```

The exact start and end of data in the body are defined by both delimiters and the `llim/rlim` pair. The last delimiter before the start address marks the start boundary, while the first delimiter following the end address marks the end boundary.

7.6.2. Reference

All references to edit-strings are via pointers to data within the body of some buffer. This differs from schemes where a reference is a more complex structure which may include a count of the remaining items in the current buffer and the address of a function to call when the count vanishes.

Data are accessed via pointers in the usual way. In the case of accessing a stream the pointer will be incremented after the access. This may take it out of the data. The test for this is performed when accessing rather than when incrementing. The condition of being out of the data is recognized when the pointer points to a delimiter and the end address given in the buffer header does not lie beyond the pointer. When storing at the end of an edit-string, as when writing a stream, the writer only need update rlim if it writes a delimiter.

7.6.3. Locating Block Headers

The question arises as to how to locate the start of the buffer given only a pointer into the buffer body. To this end all buffers are taken to be of size a power of two, and are aligned on buffer boundaries, that is, they start at an address which is a multiple of their length.

When the size of the buffer is known, the buffer header may be located by masking out the low-order bits of the pointer. In effect a pointer has two components, a pointer to the start of the buffer and an offset from that start. Unlike other ways of forming two-element structures, this approach has the benefit that the structure forms an ordinary machine address that can be used as an indirect reference and incremented in the usual ways.

When the size of the buffer is not known, it may be determined by trying all sizes in decreasing order and for each, masking out the low-order bits of the pointer as in the case when the size is known. The largest size yielding a header which (a) is within the storage region allocated to this scheme and (b) contains this size in its size field, is the size of the containing buffer. The correctness of this method depends on the observations that (a) every address within a buffer yields the buffer header using SOME mask, and (b) every mask larger than must yield some buffer header.

The advantage of this approach is that processes that know the buffer size in advance can run faster, while processes that don't, such as postmortem diagnosers, can still find their way around given only pointers into buffers.

7.6.4. Asynchronous Access

The question now arises as to asynchronous reading and writing; can a reader get a consistent picture of a string while it is being written? In general a certain amount of explicit synchronization of independent processes may be needed for some kinds of reading and writing. However for some of the cases we are particularly concerned about, explicit synchronization can be avoided.

Consider the case of any number of readers reading in either direction in a stream being written only at the ends (i.e. no inserting other than at the ends, and no deleting). By symmetry it suffices to consider a reader scanning forwards. A reader fetching a non-delimiter is assured that he is not at the end of the stream. Fetching a delimiter is more problematical; the delimiter may have been genuine at the time of the fetch, but may be replaced by some other datum by the writer before the reader fetches the end address.

If the writer has not updated the end address then no serious problem arises; the reader merely has an out-of-date picture. However suppose that between the reader's reading the delimiter and checking the end address that the writer writes a non-delimiter and then the delimiter. The writer will then bring the end address up to date and the reader will be fooled into believing that the delimiter it read is a datum.

There is an easy cure for this problem. The reader identifying a delimiter as a datum should always refetch that datum and discard the old datum. The second fetch is guaranteed to be correct since all writing is performed at the end. This method avoids the expense of explicit synchronization.

When the need does arise to synchronize explicitly, the question arises as to how to minimize overhead. System calls to raise interrupt priority or defeat the interrupt mechanisms are unduly expensive; it is considerably cheaper to rely on shared semaphores when these can be set, tested, and reset with single instructions.

References

1. F. Baskett, and A. V. Bechtolscheim. The SUN Workstation: A Hardware Overview. Stanford University Computer Science Department 1981.

2. F. Baskett, A. V. Bechtolscheim, W. I. Nowicki, and J. K. Seamons. The SUN Workstation: A Terminal System for the Stanford University Network. Stanford University Computer Science Department 1980.

3. F. Baskett, J.H. Clark, J.L. Hennessy, S.S. Owicki, and B.K. Reid. Research in VLSI systems: Design and architecture. Tech. Rept. 201, Computer Systems Laboratory, Stanford University, 1981.

4. F. Baskett, J.H. Howard, and J.T. Montague. Task communication in DEMOS. Proceedings of the Sixth Symposium on Operating Systems Principles, November, 1977, pp. 23-31. Published as *SIGOPS Operating Systems Review* 11(5).

5. D. R. Boggs, J. F. Schoch, E. A. Taft, and R. M. Metcalfe. "Pup: An Internetwork Architecture." *IEEE Transactions on Communications 28*, 4 (April 1980), 612-624.

6. S. I. Feldman. Make - A program for Maintaining Computer Programs. Part of the Unix Programmer's guide, Volume 2.

7. R. E. Gorin. Computer Networking at Stanford. Stanford University Computer Science Department 1980.

8. J.L. Hennessy. Pascal*. Tech. Rept. , Computer Systems Laboratory, Stanford University, 1980.

9. G. Kane. *68000 Microprocessor Handbook.* Osbourne/McGraw-Hill, 1981.

10. G. Kane, D. Hawkins, and L. Leventhal. *68000 Assembly Language Programming.* Osbourne/McGraw-Hill, 1981.

11. K. A. Lantz. Perseus Rising. Stanford University Computer Systems Laboratory 1980.

12. *MC68000 Design Module User's Guide.* 1979. MEX68KDM(D2).

13. *MC68000 16-bit Microprocessor User's Manual.* 1980. MC68000UM(AD2).

14. William M. Newman and Robert F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, 1979.

15. D.M. Ritchie and K. Thompson. "The UNIX time-sharing system." *Bell System Technical Journal 57*, 6 (July 1978), 1931-1946.

16. W. Yundt, Chairman. Report of the Study Group on Networking. Stanford University Center for Information Technology January 1981. Perpared for the Task Force on the Future of Computing at Stanford

Index