Linux on the Meiko CS-2*

Jim Garlick

July 5, 2002

Abstract

The Meiko CS-2 is a massively parallel processing (MPP) cluster designed for high performance scientific computing. Its compute nodes are based on the 32-bit SPARC v8 processor and are constructed with an on-board communications processor, a Hitachi H8 microcontroller, and a Control Area Nework (CAN) interface. Meiko modified the Solaris operating system to run on the CS-2. The Linux kernel was similarly modified to run on the CS-2's MK403 node. The CAN network is supported by the Linux kernel and is used as the remote system administration interface to the nodes when combined in a cluster. The Linux operating system serves as an excellent research vehicle for investigating parallel computing in general and Linux clusters in particular on Meiko CS-2 hardware.

1 Introduction

The Meiko CS-2 is a massively parallel processing (MPP) cluster designed in the early 1990's for high performance scientific computing. Though its processing elements are now obsolete, the CS-2 stands out as a well-engineered system that not only achieved outstanding performance for its era, but was designed with a level of integration sadly lacking in todays MPP's built from commodity, off-the-shelf components. Section 2 gives an overview of the CS-2 system hardware.

The CS-2 was designed to run a variant of the Solaris operating system called MSOL. Meiko is no longer in business per se and thus new versions of MSOL are not being produced. Source code for older versions is not readily available due to restrictive licensing. Nonetheless, the CS-2 hardware still has value as a research vehicle for investigating MPP system issues; therefore an open source replacement for MSOL is needed. This report describes a successful effort to port Linux to the CS-2. Section 3 describes the modifications to the Linux kernel version 2.2 necessary to boot it on the MK401 and MK403 processing elements. Section 4 explains the Control Area Network (CAN) and bargraph device drivers. Finally, section **??** suggests areas for further work.

*This work, which took place in 1998, was performed under the auspices of the U.S. Dept. of Energy at the University of California Lawrence Livermore National Laboratory under contract n. W-7405-Eng-48.

Figure 1: 224-way CS-2, Lawrence Livermore National Laboratory

2 CS-2 Overview

The Meiko CS-2 is a massively parallel processing (MPP) cluster designed in the early 1990's for high performance scientific computing. The cluster is comprised of processing elements; a high performance, low latency interconnect; storage devices; and a Control Area Network (CAN) for remote system administration and monitoring of the various components. The system is packaged in modular units: processing, peripheral, and switch modules of the same physical size are installed in CS-2 bays. A 224 processing element CS-2 is pictured in Figure 1. This machine was operated at Lawrence Livermore National Laboratory from 1995 through 1999.

2.1 Processing Elements

There exist several types of CS-2 processing elements. This report focuses on the MK403[12], based on dual Ross HyperSPARC CPU's[14], the LSI Logic SparcKit-40 family of support chips, 128 megabytes of on-board memory, and other support hardware. The MK403 was designed to be a *vector processing element*, with integrated dual Fujitsu MB92831 micro vector processors (μVP) interfaced to the system processor/memory bus (MBUS) and also directly to the memory subsystem: however, architectural limitations of the μVP's made them difficult to support in MSOL. The processors were removed in the MK403 nodes available to the author, rendering them equivalent to the less exotic MK401[11] SMP scalar processing element.

A Hitachi H8/534 microcontroller resides on the MK403 board. It provides remote reset and environmental monitoring functionality for the processing element via the CAN network and can generate a watchdog reset on the SPARC processors if programmed to do so.

The Linux port described in this report enables either the MK401 or MK403 to boot Linux. The MK402, though unavailable to the author for testing, should run the unmodified Linux kernel as it appears more like a garden variety *sun4m* workstation. The outlook for Linux on the MK405 quad processor board is unknown.

2.2 Interconnect

The CS-2 was successful in part because of its high-speed, low-latency interconnect based on the Meiko Elan communications processor[5] and Elite crosspoint switch[4]. The Elan/Elite combination achieves bidirectional bandwith of 50 megabytes/second and less than 10 microseconds latency. An Elan communications processor is built into each processing element attached directly to the 40 megahertz, 64-bit wide MBUS and can therefore access system memory as a peer with the SPARC CPU's. The Elan reduces message passing latency by allowing user level applications to perform synchronization and remote direct memory access (DMA) operations directly, bypassing the kernel. The Elan virtualizes remote DMA addresses using an on-chip memory management unit (MMU), and implements security to limit remote memory access to "exported" areas, only in the processes cooperating in a parallel job.

Each four-by-four Elite crosspoint can switch four bidirectional, byte-wide links that are clocked at 70 megahertz. The Elites are interconnected in a fat tree topology. Each stage of the switch introduces 200 nanoseconds latency. The first stages of the two planes of the CS-2 Elite switch are packaged in the *processor module*[9] that houses four processing elements; the additional stages are packaged in *switch modules*[6] that are cabled to the compute modules via flat ribbon cable built into the CS-2 bay, and to each other using external cables plugged into MD-68 connectors on the fronts of the switch modules.

2.3 Storage Devices

Peripheral modules[8] house SCSI-2 RAID arrays used in the CS-2 for parallel file system and network file system (NFS) volumes (MSOL includes a high-speed NFS filesystem that uses native Elan communications to improve performance). The peripheral modules interface to special *I/O processing elements*, typically the MK401 via external SCSI-2 MD-50 cables. A CAN interface in the module is used to monitor the RAID arrays, module power, etc.

2.4 Control Area Network

The CS-2's CAN network[10] provides a framework for remote system administration. CAN is a one megabit per second, multi-drop, serial network. A three level hierarchy was devised for the CS-2: the local CAN (LCAN) network interconnects a compute module's four processing elements, their H8 microprocessors, and the module controller board (another H8 microprocessor) internally within the module; the intermediate level CAN (XCAN) network interconnects modules via a second CAN interface on the module controller which is wired to a DB-9 connector on the back of the module; and the global CAN (GCAN) network interconnects groups of modules via a third interface on a module controller (another DB-9 on the back of the module), and is not used on smaller clusters. Meiko layers its own protocol on top of the standard CAN protocol. By the time the Meiko headers are encapsulated inside a standard CAN packet, only four of the 10 bytes in a standard CAN packet are available for payload.

2.5 Packaging

The CS-2 employs modular packaging of components to ease support and system upgrades. As alluded to above, four processing elements, the first level Elite switch, a H-8 based module controller, power, and cooling are packaged in a *processor module*. In addition, the processor module contains four hot-swappable SCSI-2 drives which are wired to the four processing elements. A CS-2 may also contain *peripheral modules* and *switch modules*. Eight modules of any type are packaged in a *bay*[7], which includes cabling for the second stage Elite switches, cooling ducts, and A.C. power distribution to the modules. A bay must be configured at the factory for a particular configuration of module types; the customer cannot easily replace compute modules with switch modules, for example. One or more bays make up a CS-2 cluster; typically bays are bolted together in groups of three. The CS-2 pictured in Figure 1 has fourteen bays.

3 Linux Port to the MK401/MK403

The Meiko MK401/MK403 nodes are similar to the Sun SparcStation 10 which runs Linux, but there are subtle differences that must be addressed in order to boot the Linux kernel on the Meiko nodes. These problems are described in detail below.

3.1 Hardware Identification

When Linux boots, it first queries the OpenBoot PROM (OBP) to discover the type of hardware it is running on, then uses this information to select hardware-dependent code paths. OBP has a environment variable called “compatible” which is set to “sun4m” on the SparcStation 10. On the MK401/403, it is set to “dino1”. Code in `arch/sparc/kernel/head.S` reads this value into the global variable `cputypval` and branches to the appropriate CPU initialization code. For dino1, the sun4m branch is taken and Meiko hardware differences are handled conditionally in the sum4m code path (if the enum `sparc_cpu_model` is set to `dino1` or `idprom->id_machtype == 0x25` then the code is executing on the MK401/403).

Meiko created a new IDPROM format to store additional fields in the IDPROM. The IDPROM version is stored in an element of the idprom structure called `id_format`. If `idprom->id_format` is set to 1, then the IDPROM is using

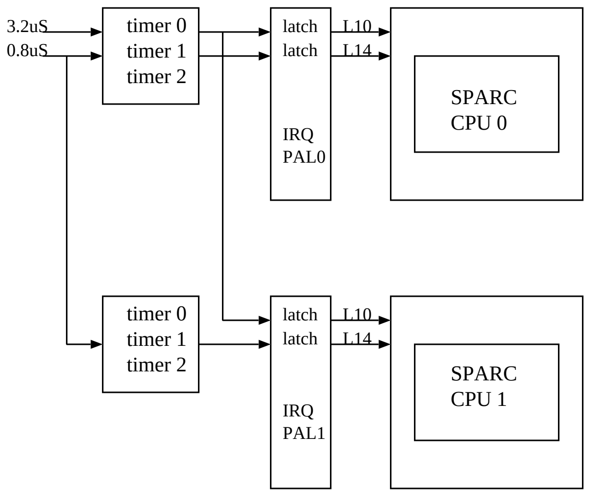

Figure 2: MK401/403 Programmable Interval Timers

the traditional version one format. On the MK401/403, the value is 2, indicating that Meiko's version two format is used. Code must be introduced to handle version two, which is the same as version one but adds two long integer fields for board revision and serial number, and subtley changes the way the checksum is computed.

3.2 Timers

Two 82C54 programmable interval timers[2] are available on the MK401/403, one assigned to each processor as depicted in Figure 2. Each 82C54 has three independent timers: CPU 0's timer 0 drives the level 10 interrupt on both SPARC's (CPU 1's timer 0 is not used); Timer 1 drives the level 14 interrupt on each SPARC; and Timer 2 is not used.

The level 10 interrupt on the SPARC is the primary mechanism used in the kernel for implementing periodic events such as process preemption and scheduling. It is meant to occur at 10 millisecond intervals. Timer 0's clock input runs at a period of 3.2 microseconds, so to get pulses occuring every 10 milliseconds on the timer output, the timer is configured in *rate generator* mode, causing it to function as a *divide-by-N* counter, and *N* (the initial count) is set to 3125.

The level 14 interrupt on the SPARC provides a periodic interrupt to the Open boot PROM (OBP) running on CPU 0, but after Linux boots, it is disabled for uniprocessor mode, and used for profiling in SMP mode. Timer 1's clock input runs at a period of 800 nanoseconds. Its output is silenced for a uniprocessor configuration by switching it to *interrupt on terminal count* mode and loading a count of zero.

Function	Description
disable_irq()	mask the specified interrupt level on current CPU
disable_pil_irq()	same as above, but call only from interrupt level
enable_irq()	enable the specified interrupt level on current CPU
enable_pil_irq()	same as above, but call only from interrupt level
clear_clock_irq()	clear a pending level 10 timer interrupt
clear_profile_irq()	clear a pending level 14 timer interrupt for specifid CPU
set_cpu_int()	send IPI to specified CPU and level (SMP only)
clear_cpu_int()	clear IPI for specified CPU and level (SMP only)
set_irq_udt()	for round-robin handling of interrupts (SMP only)
__irq_itoa()	convert interrupt level to ascii string

Table 1: Interrupt Functions Exported to Kernel

Linux uses the level 10 timer to update the *xtime* global kernel variable every 10 milliseconds. The *gettimeofday* system call reads this value and returns it to the user; however, higher resolution than 10 milliseconds is possible by reading the count on the level 10 timer, and adjusting *xtime* accordingly. In this way a *gettimeofday* resolution approaching the input clock rate of the level 10 timer (3.2 μS) can be obtained.

3.3 Interrupts

A pair of programmable logic arrays (PAL's), one per SPARC CPU, implement the MK401/403's interrupt controllers. Hardware devices are assigned fixed interrupt levels. The higher the numerical level, the higher the priority. The interrupt controllers pass the highest priority interrupt presented on the inputs through to the SPARC processor. Interrupts can be individually masked via a mask register on the controller.

The interrupt PAL's will latch interrupts generated by the programmable interval timers. The 82C54, described in Section 3.2, is not intelligent an peripheral capable of determining when it is appropriate to assert and withdraw an interrupt request. Therefore, the pulse generated when a timer in *rate generator* mode crosses the zero count must be latched until the interrupt handler has executed. The handler clears the interrupt by writing to the appropriate counter latch bit on the PAL.

Inter-processor interrupts (IPI's) are implemented by the interrupt PALs. IPI's permit one SPARC processor to generate a software interrupt on another SPARC processor in an SMP kernel.

The interrupt controller support code interfaces with the Linux kernel by exporting the functions listed in Table 1.

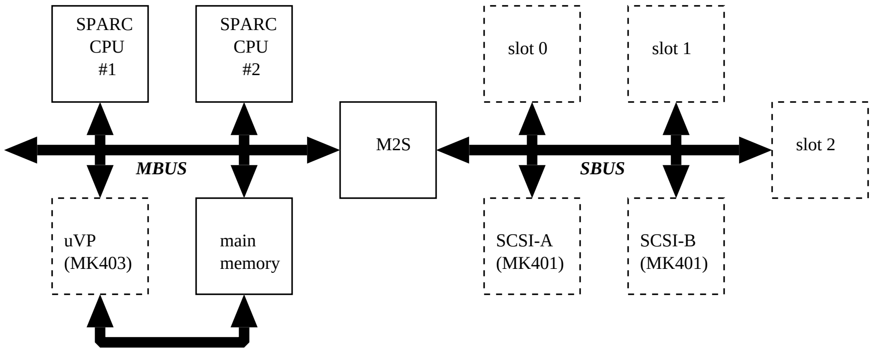

Figure 3: Bus Architecture

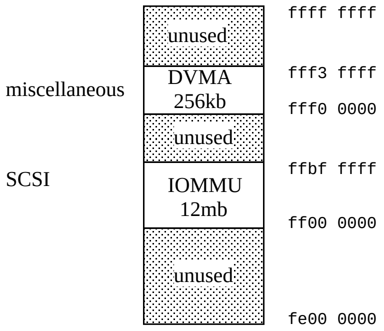

Figure 4: SPARC/Linux IOMMU Virtual Memory Map

3.4 MBus-to-SBus Controller

The 20 MHz, 32-bit SBus[1] peripheral bus is bridged to the 40 MHz, 64-bit MBus memory/processor bus with the LSI Logic L64852 MBus-to-SBus controller[3] (M2S) as shown in Figure 3. The bridging functionality requires little interaction with the Linux kernel, but kernel support is required for the on-chip input-output memory management unit (IOMMU) which translates virtual addresses generated by SBus masters to physical MBus memory addresses.

When an SBus master such as a disk controller transfers data between a device and MBus memory, this is referred to as direct virtual memory access (DVMA). Virtualizing direct memory access is beneficial because the physical pages involved need not be contiguous. Setup of a DVMA operation includes creating mappings between the virtual address range programmed into the DVMA device, and the physical addresses of the targetted MBus pages.

The IOMMU is capable of mapping the upper 32 megabytes of the SBus 32-bit virtual address space. Any page within the 36-bit MBus physical address

Function	Description
mmu_map_dma_area()	allocate and map area in *DVMA* region
mmu_get_scsi_one()	map page in *IOMMU* region
mmu_release_scsi_one()	unmap page in *IOMMU* region
mmu_get_scsi_sgl()	map set of pages in *IOMMU* region
mmu_release_scsi_sgl()	unmap set of pages in *IOMMU* region
mmu_lockarea()	unimplemented
mmu_unlockarea()	unimplemented

Table 2: IOMMU Functions Exported to Kernel

space can be mapped by the IOMMU. Linux divides the 32 megabytes of usable virtual address space into the regions shown in Figure 4. The *IOMMU* region is used for SCSI transfer buffers. The *DVMA* region is used for LANCE ring buffers and other miscellaneous mappings. The kernel calls the functions shown in Table 2 to set up and tear down mappings in these regions on demand.

The IOMMU employs a single-level linear page table in main (MBus) memory. It also implements a small translation lookaside buffer (TLB) to cache page table entries in hardware. Page table entries (PTEs) are indexed by virtual page number, where a page is four kilobytes in size. The PTE contains the physical page number and several mode bits. Linux only uses the *VALID* and *WRITE* mode bits.

The M2S does not support the Level 2 MBus cache coherencey protocol[14]; thus any MBus memory that is updated by the processor should either be marked as uncacheable or requires cache flushing at appropriate times. This includes IOMMU page tables that are stored in MBus memory.

3.5 Z8530 Serial I/O Driver

The Z8530 serial I/O (SIO) device, used for serial console I/O, requires a settling time of one microsecond between accesses. On the SPARCstation 10, this is handled in hardware, but on the MK401/403 the delay is inserted in software. [add detail after this is actually done]

4 Meiko Device Drivers

The kernel modifications described in Section 3 allow the Linux kernel to boot on the MK401/403 processing element. This section describes the Meiko bargraph and CAN drivers.

4.1 Bargraph Driver

Each processing element has access to a four-by-four array of LED's on the front of the module. The LED's are used to display power-on self test (POST) results

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12

Figure 5: Bargraph Bit Assignments

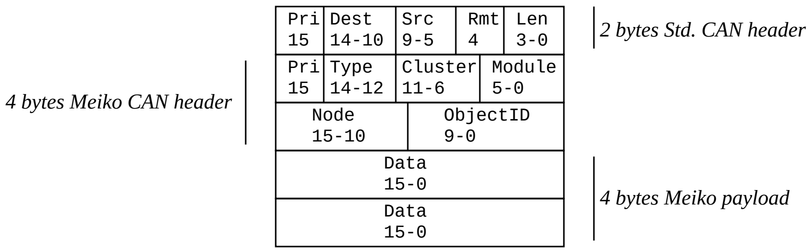

Figure 6: Meiko CAN Packet Format

when the module boots, and other status after a system boots.

The bargraph driver is almost the simplest possible Linux device driver module. The bargraph itself appears as a 16-bit latch that can be mapped into memory as the least signficant word of a 32-bit unsigned long. Bits are assigned to LED's as shown in Figure 5. To change the bargraph pattern, the 16-bit latch value is rewritten, with zeros in the lighted positions, and ones in the dark positions.

To keep things interesting and to provide visual feedback when Linux is booted, the bargraph driver registers a timer which updates an animated pattern on the display. The intent is to allow a user space process to update this pattern via bargraph *ioctl* calls. When the device is opened from user space, the default animation is suspended until close.

The bargraph registers a character device against major number 10, minor number 160. Two *ioctl* calls are implemented: *BGGET* to read an unsigned long value and *BGSET* to set an unsigned long value. The relevant definitions are available by including `<asm/meiko/bargraph.h>`.

4.2 CAN Driver

The CS-2 CAN network, briefly described in Section 2.4, interconnects all components of the CS-2 system for control and monitoring purposes. CAN is a standard one megabit per second, multidrop serial network. Meiko adds another layer to this network to implement a CAN internetwork, encapsulating

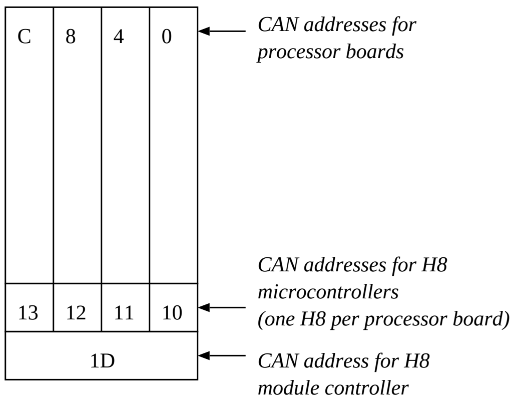
C
8
4
0
CAN addresses for processor boards
13
12
11
10
CAN addresses for H8 microcontrollers (one H8 per processor board)
1D
CAN address for H8 module controller

Figure 7: Meiko LCAN Addresses

Function	Description
can_register_svc()	register callback to receive CAN data
can_unregister_svc()	unregister callback
can_send_raw()	send CAN packet without setting remote or src fileds
can_send()	send CAN packet

Table 3: Functions Exported by can.o

their own packet in the standard CAN packet, shown in Figure 6. Meiko's CAN protocol is given a thorough treatment in [10] and is not described here. Linux implements a device driver for the standard CAN network interface (*can*). A second driver (*mcan*) that is a client of the first driver implements the Meiko CAN protcol. Two other drivers (*mcanobj* and *mcanconsole* that are clients of the second driver and implement Meiko CAN objects, one for remote console, and one for a collection of simple objects. User space code can also implement objects by interacting with the drivers via a character device interface.

4.2.1 can.o Generic CAN Driver for Philips 82C200

The MK401/403 processors use a Philips 82C200 CAN controller[13]. The 82C200 implements CAN protocol version 1.0. It has buffer space for two received packets, and can generate an interrupt on packet received, transmit buffer empty, and several error conditions. The lowest level CAN driver in Linux implements an interface for sending and receiving CAN packets. It exports the functions listed in Table 3

Kernel code wishing to make use of the CAN would normally send packets with can_send(pkt, 0), and receive them by registering a delivery callback with can_register_svc(). When a packet arrives, each registered delivery callback is called with a pointer to the new packet. Packets are not "consumed"; they are offered to all registered services. When a packet is sent, it normally goes right out on the wire, unless the chip is busy, in which case it sits in a ring buffer until the chip interrupts to indicate its transmit buffer is emtpy. Sent packets are also looped back for the benefit of promisuous mode support.

Sending can be made synchronous by calling can_send() with a nonzero second argument. This is required when, for example, acknowledging a request to halt the system (the ACK must be sent before the system is halted). The heartbeat object, described below, uses can_send_raw(), which does not set the remote or src fields in the header, to send fake "IAM" messages as though they originated from the node H8.

Note that registered service callbacks are called with locks held on can.c data structures. The service should not call back into can.c, for example to send an acknowledgement. Instead it should do that in another thread of control after the service callback has returned and dropped the locks.

In user space, a client would call open() on /dev/can, followed by ioctl(), read(), and write() calls, followed by close().

Read() returns all packets received. Each open file descriptor has its own duplicate queue of received packets. It is the job of the user code to ignore unwanted packets. Remember that some packets returned by read() may be handled by code in the kernel (see mcanobj.c). The returned value will always be a multiple of the size of can_packet_t.

Write() sends a packet, sleeping only if the output ring is full (unless O_NONBLOCK is set, in which case it returns -1 with errno set to EAGAIN). Write() sizes should always be multiples of can_packet_t.

Ioctls are available for setting and clearing promiscuous mode. There is a separate promiscuous mode bit for the chip and for each file descriptor, so while traffic on the chip receive queue will increase dramatically when any file descriptor turns on promiscuous mode, the individual file descriptor receive queues will be quiet unless their promiscuous bit is set. As noted above, transmitted packets are looped back regardless of the promiscuous bit setting.

On the Meiko, the primary client of /dev/can is "cansnoop" which optionally turns promiscuous mode on and formats and prints every packet received. Most user access to the CAN is via /dev/mcan (mcan.o), which is another module that registers itself as a kernel client of this one. It is described below.

4.2.2 mcan.o Meiko CAN Module

The mcan.o module registers itself as a client of can.o, and provides an interface to the Meiko CAN network. The Meiko CAN network, described in "Overview of the Control Area Network (CAN)" and summarized below, is a 3-level hierarchy of regular CAN networks with addressing, routing, and protocol extensions.

Meiko CAN packets are encapsulated in regular CAN packets in much the same way as IP packets are encapsulated in ethernet. Meiko consumes four bytes of the scant eight byte payload in a CAN packet to include its extended addressing and control bits, leaving four bytes for "actual" payload.

A Meiko CAN address consists of three six-bit components, one for each level of hierarchy in the network: cluster, module, and node. For intra- module communication, the node address would be the same as the CAN destination address; for inter-module or inter-cluster communication, the addresses of H8 module controllers used as routers would be the CAN destination.

Sparc nodes, their on-board H8's, and module controller H8's have a fixed CAN address within the module that is determined by their physical position. This intra-module level of network is referred to as the "LCAN" and addressing is shown in Figure **??**.

Module controllers have two CAN ports on the back whose CAN addresses are set via thumbwheel switches (power cycle if changed!). One connects to other modules in the local cluster (up to 24 modules per cluster). This level of CAN network is referred to ast he "XCAN" network. The other connects to the "GCAN" network on H8's acting as GCAN routers only. If unconnected, these ports should be properly terminated. (NOTE: GCAN has not yet been tested on linux; in theory, however, the module H8 firmware should do all the appropriate magic).

Meiko also encodes the "type" of a packet in a 3-bit field. The type may be:

```
RO - read object (ACK/NAK returned)
WO - write object (ACK/NAK returned)
WNA - write-no-ACK (nothing returned)
DAT - data (ACK/NAK returned)
ACK - positive acknowledge
NAK - negative acknowledge
SIG - signal (like a WNA broadcast)
```

Finally, packets include a 10-byte object ID. Typically objects are some 4-byte value that can be read/written via the network. For example, one can get or set the OpenBoot PROM "boot-device" option by reading/writing the BOOT_DEV object (0x3f4) on a system. A BOOT_DEV packet's payload is an integer representing the boot device, 1 for elan network, 2 for disk, 3 for network. Similarly, a write to the BREAK object (0x3f3) causes UNIX to halt, or the PROM to do the equivalent of stop-A.

One odd thing to node is that there is only room for one Meiko extended address per packet. For RO/WO/WNA/DAT/SIG packets, it contains the destination address. For ACK/NAK packets, it contains the source address. When a reply is sent, the regular CAN source and destination fields are swapped, and if the destination happens to be a module H8 (router), the H8 uses state information recorded when the request was originally sent to route the packet accordingly.

4.2.3 mcanobj.o Meiko CAN Object Dispatcher

4.2.4 mcanhb.o Meiko CAN Heartbeat Module

4.2.5 mcantty.o Meiko CAN TTY Module

4.2.6 mcancon.o Meiko CAN Console Module

These sections were never completed.

5 Acknowledgements

Thanks to David Hewson, Moray McLaren, and Duncan Roweth, formerly of Meiko, Ltd., for answering some key questions about the hardware.

A special thanks to Fred Miller, Livermore's "meikotag repairman", for his tireless support and patience with the author's sometimes ill-conceived hardware experiments.

References

[1] Ben J. Catanzaro. *The SPARC Technical Papers.* Springer-Verlag, 1991.

[2] Harris Semiconductor. *82C54 CMOS Programmable Interval Timer*, March 1997.

[3] LSI Logic. *L64852 MBus-to-SBus Controller (M2S) Technical Manual*, 1992.

[4] Meiko Ltd. *Overview of the CS-2 Communications Network*, 1993.

[5] Meiko Ltd. *Overview of the CS-2 Communications Processor*, 1993.

[6] Meiko Ltd. *The CS-2 Switch Module*, 1994.

[7] Meiko Ltd. *The CS-2 Bay*, 1995.

[8] Meiko Ltd. *The CS-2 Peripheral Module*, 1995.

[9] Meiko Ltd. *The CS-2 Processor Module*, 1995.

[10] Meiko Ltd. *Overview of the Control Area Network (CAN)*, 1995.

[11] Meiko Ltd. *SPARC/IO Processing Element (MK401) Users Guide*, 1995.

[12] Meiko Ltd. *Vector Processing Element (MK403) Users Guide*, 1995.

[13] Philips. *PCA82C200 Stand-alone CAN Controller*, 1990.

[14] Ross Technology, Inc. *SPARC RISC User's Guide, HyperSPARC Edition*, September 1993.