270010 DATAMOVER

HARDWARE MANUAL

68000 MICROPROCESSOR BOARD

256 KBYTE RAM

JUNE, 1982

REV. C

TABLE OF CONTENTS

1. INTRODUCTION AND INSTALLATION

1.1 INTRODUCTION

The Datamover-256 is an advanced high speed microprocessor/memory expansion for the MTU-130 desktop computer. In use it provides the MTU-130 user with 256K additional bytes of memory, all fully accessible by the MTU-130's 6502 processor. It also provides a slave processor function using the Motorola MC68000 16 bit microprocessor which is currently the most powerful microprocessor in general use. The 68000 can directly access the 256K of memory as well. Together these Datamover features allow the MTU-130 user to tackle applications heretofore impossible or impractical to do on a low cost desktop system.

This manual fully describes the hardware of the Datamover board as well as programming techniques peculiar to the Datamover. Separate manuals describe the MC68000 instruction set, Datamover cross monitor and assembler, Datamover BASIC, and other available Datamover software products.

1.1.1 The Slave Processor Concept

A slave processor is, literally, an additional microprocessor (computer chip) that is controlled by the master (main) system microprocessor. Slave processors may be added to a system to split up the total system workload, to execute certain parts of a program faster, or to easily upgrade to a more powerful processor. The major advantages of using a slave processor (as opposed to replacing the main processor which is not generally possible) are preservation of investment, low incremental cost, and often better performance. Of these, preservation of investment is perhaps the most important since current system hardware, software, and use experience may continue to be utilized.

A slave processor generally has direct access only to some quantity of its own memory and an interface to the main processor. The master processor has direct access to all system resources as usual. Thus the slave must access other system resources by communicating with the master. This two-step access to resources such as the disk and display can generally be made transparent to the system user, however.

1.1.2 Advantages of the MC68000 as a Slave Processor

The system designer has many options available for slave processors. High speed arithmetic processors for example can substantially simplify and speed up math intensive programming of the master processor. The use of a general purpose 16 bit microprocessor as the slave however allows a much broader application range than a special purpose processor would. Even if usage were restricted to math, its programmable nature makes emulating current math routines (such as in MTU BASIC) easier and allows optimization for special applications. Other advantages include the possibility of running large programs (such as languages) on the slave itself thus giving the illusion that it is really the master processor.

Below are a few of the factors that were considered when selecting the 68000 as the processor for the Datamover:

1. Reasonable microprocessor chip cost and ready availability.
2. Ability to directly access very large amounts of memory.
3. Overall high speed, particularly in mathematics which makes it as fast as the most common math processors.
4. Relatively simple interface logic in a small system.

Following is a summary of the Datamover-256 specifications. Detailed descriptions of these and other points may be found elsewhere in this manual.

1. Memory Capacity - 256K bytes RAM on-board expandable to 1024K with a piggyback board. Type 4164 or equivalent RAM ICs operated with a 375NS cycle time.

2. Microprocessor - Motorola MC68000L8 or equivalent operated at 8MHz.

3. Architecture - Slave processor with dual-port memory plus bi-directional interrupts. All memory accessible from either port.

4. MC68000 Port - Organized as 128K (expandable to 512K) 16 bit words, zero wait state access unless pre-empted by MTU-130 access or a refresh cycle. Guaranteed worst case access time is less than 2uS, guaranteed worst case cycle rate is .93MHz when the MTU-130 bus port is 100% utilized (1.0MHz cycle rate).

5. MTU-130 Bus Port - Organized as two (expandable to 8) groups of 128K bytes each. The current 128K appears in banks 2 and 3 of the MTU-130 address space. Guaranteed zero wait state access. The bus port operates at 1MHz.

6. Refresh - Distributed refresh with a cycle taken each 16uS. Refresh cycles refresh all RAM ICs at once whereas normal read and write cycles only activate the addressed row of 16 RAM ICs.

7. Memory Contention Overhead - The 68000 runs at 98.4% of theoretical 8MHz zero wait state speed when the MTU-130 bus port is not addressed, typically greater than 90% when accessed by the MTU-130 as a data bank, typically greater than 65% when accessed by the MTU-130 as a program bank, and guaranteed greater than 46% with 100% usage of the MTU-130 bus port.

8. Datamover Control - The 68000 may be halted and reset from the MTU-130 port under program control as well as by a general MTU-130 reset. The halt/run status of the 68000 may be read from the MTU-130 port.

9. Interrupts to the 68000 - The MTU-130 port may independently initiate interrupts on levels 4 and 7 and determine when level 4 is acknowledged by the 68000. Autovectoring to locations 000070 and 00007C respectively is used.

10. Interrupts to the 6502 - The 68000 may set an interrupt on the MTU-130 bus and determine when it is cleared. The MTU-130 may enable or disable such interrupts from the 68000.

11. Special Features - Memory expansion connector containing multiplexed addresses, parallel 16 bit data, timing signals, and decoding for 7 rows of 64K words each (458K). Alternate address jumper and programmable board enable to allow 2 Datamovers in the same system.

12. Exceptions - Bus errors are not possible. There is no trap for addressing non-existent memory. There is no lockout of MTU-130 port accesses during execution of the Test and Set (TAS) instruction.

13. Power Consumption - +8 volts unregulated at 1.0 amp with the 68000 running.

14. Physical Size - The Datamover board measures 11" wide by 5" deep. Maximum component height is 0.5".

1.2 INSTALLATION

Installation of the Datamover-256 involves removing the top cover of the MTU-130 keyboard unit, removing the RF interference shield, inserting the Datamover into an empty slot, and then reassembling the keyboard unit. After installation, a test program is run which is included on the software disk that was packed with the Datamover.

1.2.1 Disassembly Instructions

1. Unplug all cables from the MTU-130 keyboard unit and move it to an uncluttered, well lit work area.

2. Carefully stand the keyboard unit on its right end with the fan down.

3. Locate then remove two rear cover screws which are nearest the two back corners of the bottom plate and symmetrically placed. Also remove two front cover screws which are nearest the two front corners and go through long slotted holes in the bottom plate.

4. Lay the unit back down in its normal operating position but with the front overhanging the table by a couple of inches.

5. Remove the 4 screws along the front bottom surface that fasten the top cover to the bottom plate. Be sure to save the washers if present.

6. Slide the top cover forward about 1/2 inch then lift the front while keeping the back stationary so that it effectively "hinges" up about 40 degrees.

7. On the left side of the cover, locate the 2-wire cable going to the speaker and pull off the quick-disconnect lugs from the speaker terminals.

8. Lift the cover off completely and stand it upright at the right side of the base. The keyboard cable will be removed later. The fan wires may remain connected to the power supply board.

9. Remove the R-F shield which covers the front part of the boards. There are two screws on each edge of the shield.

10. Gently unplug the keyboard cable from its connector on the bottom PC board (use a small screwdriver to pry it up).

11. You should see the Monomeg CPU board (the large one in the bottom slot) and the Disk Controller board plugged into the card file. These need not be disturbed when installing the Datamover.

1.2.2 Handling and Installing the Datamover

The Datamover board uses a number of MOS ICs in its construction. These are more sensitive to damage from static electricity than are TTL ICs which is why the Datamover is shipped in a black, conductive plastic bag. When handling the Datamover outside of this bag you should always hold it by the large black finned heatsink. If it must be set down temporarily, sit it on top of its bag flattened out. If it is being set on a metal table or other highly conductive surface, touch the surface first with your hand to equalize the charge between your body (and the board since you are holding it) and the surface.

The Datamover may be installed into any empty slot in the MTU-130's card file. Unless the boards have been moved, there will be an empty slot for large boards between the Monomeg CPU board and the Disk Controller board. There will also be two empty slots for smaller boards above the Disk Controller. Since the Datamover is a small (5" deep) board, it is desirable that it be installed in one of the top two slots. If this is the only expansion board in your MTU-130 (i.e., a total of 3 boards in the card file), it is desirable that it be installed in the top slot for maximum ventilation clearance.

To install the Datamover, remove it from its black bag and hold it by the heatsink. Before plugging it into a slot, touch one of the heatsinks on the MTU-130 power supply to discharge your body. Then slide the board into the card guides and firmly seat its edge fingers into the socket by applying pressure with the heels of both hands. There should be a definite final movement when the edge fingers seat into the socket. Make sure the board is straight in its socket and that each edge is in its white plastic card guide slot.

1.2.3 Reassembly Instructions

1. Find the top cover and re-attach the keyboard cable to the Monomeg CPU board.

2. Find the R-F shield and install it with its 4 screws. Insert each screw only part way until all 4 have been inserted. Then push the shield up and back as far as it will go and tighten the screws.

3. Position the top cover in the "hinged up" position it was in step 6 earlier and reattach the two speaker lugs.

4. Lower the cover and slide it back until the top rear edge is flush with the back panel. The short tabs under the rear of the top panel should hook under the lip of the back panel. It may be necessary to press the top panel flat while doing this.

5. While holding the two halves together with your hands, stand the unit on its right end like it was in step 2 of the disassembly instructions.

6. Carefully manipulate the cover as necessary to make the rear corner holes in the bottom plate match up with the internal cover mounting brackets and install the two rear corner screws (these were the ones that were removed in disassembly step 3). Also install the two front corner screws.

7. Set the unit back down with the front overhanging the tabletop edge and reinstall the 4 front cover screws that were removed in disassembly step 5.

8. Reconnect the keyboard unit to the disk drives, monitor, and power cord.

1.3 RUNNING THE DATAMOVER TEST PROGRAMS

Before specifically testing the Datamover, obtain a known good working disk and start up the MTU-130 as usual. There should be no perceptible difference in its operation. If the system fails to function normally, turn the power off and refer to the Troubleshooting section of this manual for advice.

The Datamover distribution disk has two diagnostic programs and possibly other Datamover support software recorded on it as well as essential CODOS system files. It is recommended that you copy all of the files from this disk to one of your working disks by using the COPYF command with no arguments (the operating system files will already exist so they will not be copied). Single drive users should copy all unfamiliar file names seen on a files listing of the Datamover disk. After copying, put the Datamover distribution disk in a safe place.

The first diagnostic program performs a functional test of the Datamover's circuitry. It checks the memory access logic, run/halt/reset logic, the bidirectional interrupts, and verifies that the 68000 can run a simple program. Assuming that a disk with the Datamover files is in drive 0, enter the following command to run this program:

```
DMFTEST
```

The program should print a message as each functional area of the board is tested indicating that it is OK. If a problem is found, refer to the Troubleshooting section for advice. Note that only memory groups 0 and 1 are functional on a standard 256K Datamover.

The second diagnostic program performs a comprehensive test of the Datamover's memory. Assuming that a disk with the Datamover files is in drive 0, enter the following command to run this test program:

```
DMMTEST
```

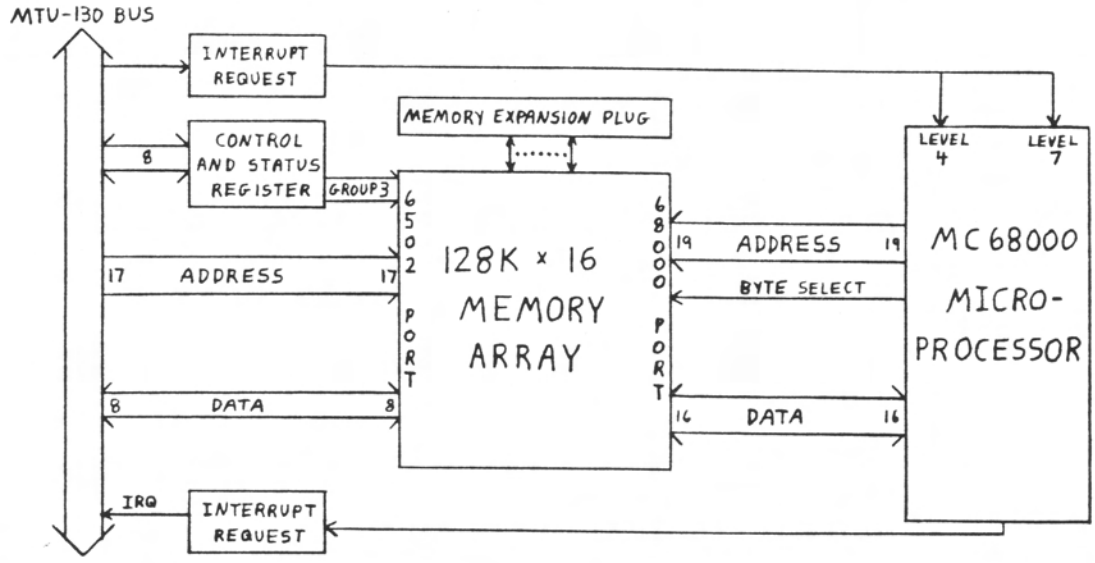
To make this test more meaningful, the MTU-130's 6502 processor will test half of the memory (the upper 128K) while the 68000 simultaneously tests the other half. As each processor completes a pass, it will print a message in the upper portion of the screen indicating the pass number. If either processor detects an error, the error information will be printed in the lower portion of the screen. Both processors will stop when the lower part of the screen fills. The test algorithm consists of storing random numbers in the block to be tested and then reading them back. Every 16 passes there is a 15 second delay inserted between the store and readback phase to check the memory refresh logic. Note that the 68000 performs a test pass in about 1/7 the time required by the 6502. The long term ratio however is about 6-to-1 since a 15 second delay takes the same amount of time on either processor.

2. PROGRAMMING THE DATAMOVER

This section deals with programming the interface between the MTU-130's 6502 microprocessor and the memory/68000 microprocessor on-board the Datamover. Programming of the 68000 itself will not be covered here. If you plan to use the Datamover strictly as a memory board (or you purchased it less the 68000 chip), then only the Enable Register description in section 2.2.3 is applicable.

2.1 DATAMOVER ARCHITECTURE

Below is an illustration the fundamental architecture of the Datamover. Essentially it is a 2-port memory with an MC68000 connected to one port and the MTU-130 bus with its 6502 processor connected to the other port. The two processors may also communicate by interrupting each other. Finally, the 6502 exercises ultimate control by being able to halt and reset the 68000.



2.1.1 Shared Memory

The majority of the board's logic implements a 2-port memory. Its size is normally 256K bytes but could be as little as 128K or as much as 1024K. Unlike usual microprocessor memory arrays, the Datamover's memory has two functionally independent access ports. Each access port implements a complete random access read/write memory function and each can reach all parts of the memory array.

The two ports are not identical however. The 6502 port is byte-addressed and is only 8 bits wide. Bank switching logic is included to allow the 6502 to access all of the possible addresses through a 128K "window" in the MTU-130's address space. The 68000 port is word-addressed and is 16 bits wide. Since the 68000 has 23 address lines, it can address all of the memory (even the 1024K version) directly without bank switching. Note that it is logic inside the 68000 that splits the 16 bit words into 8 bit bytes when called for by an instruction.

Since the two ports refer to the same memory cells, it follows that what is written through one port can be read through the other and vice-versa. This of course is fundamental since it allows the 6502 and CODOS to load a 68000 program and data into the memory and conversely allows the 6502 to read back the results. The relation between the ports is such that a given 6502 port byte address will refer to the same memory cells as the exact same 68000 port byte address.

Although the two ports are functionally and logically independent they are not physically independent. What this means is that the memory cells cannot do two operations (i.e., respond to the two ports) simultaneously. Thus if one port is busy accessing the memory cells, the other will have to await its turn. In nearly all cases, the 68000 will use its port much more than the 6502 will. In these cases the 68000 will seldom have to wait for access and so its execution speed will closely approach the theoretical maximum. It is possible however to tie up the 6502 port and thus significantly slow down the 68000. This would occur if a 6502 program were loaded into the Datamover memory and executed there (i.e., the program bank were set to 2 or 3). In such a case, the 68000 could be slowed to as little as 1/2 of its normal speed although 2/3 is more typical. On the other hand, occasional (such as every 20uS) 6502 accesses for data (data bank set to 2 or 3) would slow the 68000 by only a few percent.

The Datamover memory is dynamic and therefore must be refreshed to retain its data. Logic on-board does this automatically and need not concern the programmer or user. Refreshing does however occasionally introduce a short wait in the 68000 port. The frequency and duration of these waits is such that the 68000 runs at about 98.4% of its theoretical maximum speed (assuming no use of the 6502 port). Thus the running time of a 68000 program is subject to a slight amount of uncertainty. The timing design is such that the 6502 port never waits on refresh.

2.1.2 Reset and Halt of the 68000

In addition to the MTU-130's 128K window into Datamover memory, there are three addressable registers that are in the 130's I/O address space. These registers allow the 6502 to control the 68000's "front panel". The most fundamental panel operations are Reset and Halt/run.

The Reset function does just what its name implies - it instantly halts the 68000 program and forces the 68000 and its interface circuitry into a known and idle state. This Reset signal is actually generated by an interface register bit. The 6502 can freely set this bit to a zero (forces and holds 68000 reset) or set it to a one (allows the 68000 to start and run a program). Pressing the Reset key on the MTU-130 console also forces the reset bit to a zero and thus resets the 68000 as well as the 6502. When the Reset signal is released (6502 writes a one into the interface register bit), the 68000 goes through a well-defined startup sequence and then starts executing its program. Thus to load and run a 68000 program, you would write a zero into the Reset bit, load the program into Datamover memory, and then write a one into the Reset bit to start execution. This sequence is explained in more detail in section 2.4.

A Halt function is also provided. It differs from Reset in that Halt merely stops the 68000 program while Halt is active. When Halt is released, the program continues with its next instruction. Reset on the other hand always starts the program from the beginning when it is released. Like Reset, Halt is an interface register bit and can be set to zero (halt) or one (run) by the 6502. Thus the 6502 can halt a 68000 program at any time and then restart it later. The 68000 can also halt itself by executing a Halt instruction. This halt status is logical-ORed with the Halt signal from the interface register and the 6502 can read the result from an interface status register. Thus the 6502 can tell if the 68000 program has halted itself.

The Motorola manual indicates that by pulsing Halt, single-step execution of a 68000 program can be accomplished. However, the 68000 is so fast that it may actually execute several instructions in the time required by the 6502 to turn the bit on then back off. Thus best use of the Halt function is to halt a runaway 68000 program with the option of letting it resume later (such as after examining memory to see what was happening).

2.1.3 Datamover Interrupts

While the shared memory architecture of the Datamover allows highly efficient exchange of data between the 6502 and the 68000, it is not very good for exchanging "event signals" such as "I have finished this task", or "Here's some data", or "Tell me when you're finished". While there are ways of implementing such signalling solely through "mailboxes" in memory, the Datamover also offers bidirectional interrupts to accomplish this.

If the 6502 wishes to signal a running 68000 program that some more data is available, it can store a message in memory and then interrupt the 68000 to tell it to read the message. The 6502 knows that the message was received when the 68000 clears the interrupt. Likewise, the 68000 can signal the 6502 that it is done by storing a message and then interrupting the 6502. It too can tell that the message was received when the 6502 clears the interrupt. The advantage of using interrupts for signaling is that both processors can be "doing something useful" rather than sitting in a loop testing the "mailbox" location.

It should be noted that the 68000's "Test and Set" instruction, which is intended to be used for in-memory mailbox signaling, should not be used for that purpose. The reason is that there is no "lockout" of 6502 memory accesses between the read and write portions of that instruction's execution cycle. Besides, the 6502 has no equivalent instruction for implementing such an exchange in the opposite direction.

2.1.4 Interrupts from 6502 to 68000

The MTU-130's 6502 microprocessor can interrupt the 68000 on two different priority levels. The level 7 interrupt to the 68000 is the highest possible priority and therefore cannot be locked out by the 68000 program (unless of course it is halted). It thus acts like the "non maskable" interrupt of the 6502. The principal use of this interrupt would be in conjunction with a "cross monitor" program which would allow the user (through the 6502) to unconditionally interrupt the 68000 and find out where it was executing and what the registers are. This is analogous to the INT key on the MTU-130 keyboard which unconditionally interrupts the 6502. The level 7 interrupt may be set by the 6502 writing a one into an interface register bit. The 68000 usually acknowledges a level 7 interrupt instantly unless it has been halted by the 6502 or a multiply or divide instruction is in progress.

The level 4 interrupt to the 68000 is just like the level 7 interrupt except that the 68000 program may disable it by setting the current priority level higher than 4. Thus depending on the 68000 program, a level 4 interrupt from the 6502 may not be acknowledged immediately, if ever. This is analogous to the maskable interrupt of the 6502 which can be masked off by the program. In addition, the 6502 can determine when the 68000 acknowledges the level 4 interrupt by reading an interface status register.

The two interrupts from the 6502 to the 68000 are implemented as autovector interrupts. This means that the address of the 68000 service routine for level 4 should be stored in Datamover locations $000070-000073 and the address for level 7 should be stored in locations $00007C-$00007F. Note that all addresses in the 68000 are stored in natural order (most significant byte first) so for example, if the level 7 service routine is located at $014530, $00 should be in $00007C, $01 in $00007D, $45 in $00007E, and $30 in $00007F.

The sequence for using the level 4 6502-to-68000 interrupt would be as follows:

1. The 6502 requests a level 4 interrupt by writing to an interface register.

2. Assuming that the 68000 interrupt level mask is 4 or lower, the 68000 will be interrupted and will go through its interrupt response sequence eventually ending up in the service routine for 6502 interrupts.

3. Simultaneous with entering the service routine, the level 4 interrupt is cleared. The 6502 can determine when this occurs by reading an interface status register.

4. The 68000 service routine performs the appropriate processing such as reading a message in memory.

The 6502 only knows when the 68000 service routine starts, not when it has been completed. Unless the service routine is very long, it will probably finish before the 6502 can do anything significant anyway. The sequence for the level 7 interrupt is similar except that there is no significant service delay and therefore there is no provision for the 6502 to determine when the level 7 interrupt has been acknowledged.

Note that resetting the 68000 will also clear both interrupt requests. Since resetting the MTU-130 also resets the 68000, that too will clear both interrupt requests.

2.1.5 Interrupts from 68000 to 6502

The 68000 can also interrupt the 6502 but only on the maskable interrupt level. The least significant bit of the byte at 68000 address $100001 is the interrupt request flag. The 68000 can freely set and reset it by writing (1=request interrupt) and can determine its current state by reading. The 6502 can only reset it by writing to an interface register. The 6502 can choose to ignore this interrupt in two ways. First it may mask off all interrupts by setting its interrupt disable flag with the SEI instruction. Or it may just mask off the Datamover interrupt by setting an interface register bit.

The sequence for using the 68000-to-6502 interrupt would be as follows:

1. The 68000 requests an interrupt by writing $01 to location $100001.

2. Assuming that the 6502 interrupt disable flag is off and interrupts from the Datamover are enabled, the 6502 will be interrupted and will go through its interrupt response sequence eventually ending up in the service routine.

3. The 6502 interrupt service routine checks all possible interrupt sources and determines that the Datamover is the cause of the interrupt.

4. The 6502 service routine performs the appropriate processing such as reading a message in Datamover memory.

5. When the 6502 service routine is finished processing the 68000 interrupt, it resets the request by writing to an interface register.

6. The 68000 can determine when the 6502 has acknowledged the interrupt by reading location $100001 and waiting for bit 0 to go to a 0.

Note that resetting the 68000 will also clear the interrupt request. Since resetting the MTU-130 also resets the 68000, that too will clear the interrupt request.

2.2 6502 ADDRESSABLE INTERFACE REGISTERS

The Datamover has 3 interface registers that are addressable by the 6502. Two are write-only, one is read-only, and together they occupy 2 I/O addresses; $BFBE and $BFBF in bank 0. These addresses may be changed to $BFBC and $BFBD by moving a jumper as described in section 2.5. The bit assignments and functions of these registers are described in the following sections:

2.2.1 Control Register $BFBE

The Control Register is a write-only register addressable by the 6502 at $BFBE. Individual bits control most of the Datamover's functions. This register is cleared to zeroes whenever the MTU-130 is reset. The function of each bit is described below:

Bit 7 Interrupt from 68000 acknowledge - Writing a one into this bit will clear the interrupt request from the 68000. Writing a zero will do nothing.

Bit 6 Interrupt to 68000, level 4 - Writing a one into this bit will set the level 4 interrupt request to the 68000. Writing a zero will do nothing. Once set, this bit may be cleared only by the 68000 acknowledging the interrupt or by resetting the 68000.

Bit 5 Enable interrupts from the 68000 - Writing a one into this bit will allow an interrupt request from the 68000 onto the 6502's bus. Writing a zero will prevent interrupt requests from the 68000 from affecting the 6502 bus.

Bit 4 Interrupt to 68000, level 7 - Writing a one into this bit will set the level 7 interrupt request to the 68000. Writing a zero will do nothing. Once set, this bit may be cleared only by the 68000 acknowledging the interrupt or by resetting the 68000. Since this interrupt level is non-maskable by the 68000, it will always be acknowledged immediately. Thus a separate interrupt is generated each time a 1 is written into this bit.

Bit 3 $\overline{\text{Halt 68000}}$ - Writing a zero into this bit will halt the 68000 and keep it halted until a one is written. The 68000 may also halt itself in which case it may be restarted only by being interrupted or reset while this bit has been set to a one.

Bit 2 $\overline{\text{Reset 68000}}$ - Writing a zero into this bit will reset the 68000 and keep it reset until a one is written. To perform a proper reset, the Halt bit (bit 3) should also be simultaneously set to zero. The 68000 will perform its initialization sequence and start running when ones are simultaneously written into bits 2 and 3.

Bit 1 Right Byte First - Writing a zero into this bit selects normal operation of the 6502 memory port in which even addressed bytes appear leftmost in the 68000's 16 bit word. Writing a one into this bit selects reversed operation where even addressed bytes appear rightmost in the word. The normal setting is zero; the reversed setting is used for special applications where the 6502 wishes to pass data with the least significant byte first to the 68000 which expects the most significant byte to be first in a 16 bit word.

Bit 0 --Spare--

2.2.2 Status Register

The Status Register is a read-only register addressable by the 6502 at $BFBE. Individual bits in this register reveal the complete status of the Datamover. Most of these bits are in reality readbacks from selected bits in the write-only Control and Enable registers. The status register may be read as often as desired; the act of reading it does not clear or trigger anything. Since an MTU-130 reset clears everything on the Datamover, the Status Register will be all zeroes after a reset. Below is listed the meaning of each Status Register bit:

Bit 7 Interrupt request from 68000 - This bit is a one when the 68000 is trying to interrupt the 6502. It is actually a copy of the 68000 addressable 6502 Interrupt Request bit (see section 2.3).

Bit 6 Interrupt request to 68000, level 4 - This bit is a one when a 68000 level 4 interrupt is being requested. It returns to a zero when the 68000 acknowledges the level 4 interrupt. This bit is actually a copy of Control Register bit 6.

Bit 5 Enable interrupts from 68000 - This bit is a copy of Control Register bit 5.

Bit 4 68000 Halt signal - This bit is a zero when the 68000 is halted and a one when it is running. It is the logical-AND of Control Register bit 3 and the internal 68000 Halt signal.

Bit 3 Readback of Board Enable - This bit is a copy of Enable Register bit 3.

Bit 2 Readback of Group Select 4 - This bit is a copy of Enable Register bit 2.

Bit 1 Readback of Group Select 2 - This bit is a copy of Enable Register bit 1.

Bit 0 Readback of Group Select 1 - This bit is a copy of Enable Register bit 0.

2.2.3 Enable Register

The Enable Register is a write-only register used to control the 6502 memory access port on the Datamover. Through it the access port can be disabled (disconnected from the MTU-130 bus) and different 128K memory "groups" may be made current and thus accessible to the 6502. This register is cleared to zero by a system reset which enables the access port and selects memory Group 0.

Bit 7-4 Not used

Bit 3 Board Enable - Writing a zero into this bit will enable the 6502 access port into the Datamover's memory. Writing a one will disable the memory but will have no effect on access to the Control, Status, or Enable registers. This function facilitates the use of multiple Datamovers in the same MTU-130.

Bit 2 Group Select 4
Bit 1 Group Select 2
Bit 0 Group Select 1

These 3 bits select a particular 128K memory "group" number (0-7) accessible to the 6502 port. The standard Datamover only has Group 0 ($000000-$01FFFF) and Group 1 ($020000-$03FFFF), but an expanded Datamover may have all 8 groups implemented. Group selection only applies to the 6502 port; the 68000 may access the entire memory all at once.

2.3 68000 ADDRESSABLE INTERFACE REGISTER

The Datamover has just one interface register that is addressable by the 68000. In addition only one bit of this register is significant. The register is located at $100001 in 68000 address space and the significant bit is 0. This is a read/write bit which may also be reset to zero by the 6502. Writing a one into this bit (typically by a BSET instruction) will request an interrupt on the 6502 bus (assuming that Datamover interrupts have been enabled by the 6502). When the 6502 interrupt service routine has completed its task, it will clear this bit. The 68000 may also clear the bit with a BCLR instruction if it desires and check on its status with a BTST instruction. Resetting the 68000 will also cause this bit to be reset to zero.

2.4 LOADING AND RUNNING A 68000 PROGRAM

Because of the great flexibility of CODOS and straightforward implementation of the Datamover, no specialized software is necessary for loading and running a 68000 program. You can actually set things up so that by simply typing in the program name to CODOS, the 68000 program will be loaded, an interface program on the 6502 will be loaded, and the two of them will start execution. From the program user's point of view, it would be no different from running a regular 6502 program! The sections below describe how this may be done.

2.4.1 Steps in Running a 68000 Program

In order to load and run a 68000 program, the following functions must be provided for in some manner:

1. The Datamover should be reset to an idle condition because the 68000 may wipe out the new program if it is running. This is easily accomplished by writing $00 into locations $BFBE and $BFBF.

2. The code of the 68000 program must be loaded into Datamover memory. Step 1 above made the lowest 128K of Datamover memory addressable in banks 2 and 3. Thus loading the program is simply a matter of using a CODOS GET command to load it.

3. The 68000's reset vector (locations $000000-000007 in Datamover memory) must be set. Locations 0-3 is the initial stack pointer value and locations 4-7 point to the program origin. These 8 locations are addressable by CODOS at 0:2 - 7 and can be loaded as part of the GET in step 2 or set manually with a SET command.

4. In most cases, you will need some regular 6502 code to communicate with the 68000 program and thus give it virtual access to CODOS and MTU-130 I/O devices. This code can also be loaded with a GET command. It is possible to debug a 68000 program without an interface program by simply looking at the results in memory with regular CODOS DUMP commands. Note that you can freely do this even while the 68000 is running!

5. The last step is to start the 68000 running. The best way is to have code in the step 4 interface program write $0C into location $BFBE which will start the 68000 after the interface program has started. If there is no interface program, you may start the 68000 by manually setting location $BFBE to $0C (ignore the memory verify failure CODOS message).

2.4.2 Setting Up a Load-and-Go 68000 Program

The following steps may be followed for creating a "load-and-go" 68000 program file that the user can run merely by typing in the program name. The two diagnostic programs on the Datamover distribution disk (DMFTEST and DMMTEST) were prepared in this fashion and should serve as examples.

1. Load the 68000 program and any associated data into Datamover memory. For illustration we will assume that the program occupies 68000 locations $000400-$00623F and has a data table from $014000-$017FFF.

2. SET 68000 locations $000000-$000007 with the initial stack pointer and program entry point. For illustration, assume that the stack pointer should be $008000 and the entry point is $000400. Thus you should enter:

   SET 0:2 00 00 80 00 00 00 04 00.

3. Load the 6502 interface program into memory. For illustration we will assume that it will start up the 68000 itself, it resides at $700-$B3F, and its entry point is $700.

4. Set a pair of zeroes in some unused memory locations. For illustration we will use a SET 0 0 0 to accomplish this.

5. Now that everything has been set up in memory, a single CODOS SAVE command can be used to create the load-and-go file. For the illustration case, the command would be as follows:

   SAVE SUPERDO 700 B3F 0=BFBE 1 0:2 7 400:2 623F 4000:3 7FFF

The first block saved is the 6502 interface program with its entry point of $700. The second block saved is the pair of zeroes which when the file is reloaded will reset the Datamover. The third block is the 68000 reset vector and the fourth block is the 68000 program itself. The last block is the data table for the 68000 program which is in bank 3.

When the program is reloaded and run by typing in SUPERDO in response to the CODOS prompt, all 5 blocks are reloaded and then the interface program is started at its entry point of $700. It in turn starts up the 68000 at the appropriate time by writing $0C into location $BFBE. Try doing a GETLOC on the two diagnostic programs for additional examples.

2.5 USING ASM TO PREPARE 68000 PROGRAMS

For preparing large 68000 programs one would obviously want to use a 68000 assembler (either cross or native) or perhaps a high level language. For doing smaller programs without these extra cost items, the MTU standard 6502 assembler can aid the "hand assembly" process considerably. For example, the two 68000 diagnostic programs were done in this manner and the source files supplied on the Datamover distribution disk can serve as illustrations.

A typical microprocessor machine instruction consists of from 1 to 3 parts. All instructions have an operation code (op code) that specifies what to do and what registers to act on. Most also have a single operand which is either a memory address or the data itself. Some processors have instructions with two operands. When using a non-native assembler to assemble the machine's code, the strategy is to "assemble" each of these parts separately as .BYTE or .DBYTE statements and then string them together to form whole instructions. Typically for op codes, each of the target machine's mnemonics is first equated to a bit pattern. Then by writing something like: .DBYTE NBCD the bit pattern for the NBCD instruction's opcode is generated. The operands are even simpler usually being just additional expressions following the op code. You can of course use symbols for addresses and data in these expressions as well as explicit numbers.

The big advantage of this over hand assembling code on paper is that by using symbolic addresses in the .BYTE and .DBYTE statements, you can let the host assembler calculate the actual addresses when the program is assembled. Thus you may insert and delete instructions, move blocks of code, specify relative addresses, and equate program parameters to symbols just as easily as if the native assembler was available. The following sections will explain how to quickly "hand assemble" 68000 programs using the standard MTU 6502 assembler.

2.5.1 Using Equates for the Op Codes

Although the 68000 is advertised as having only 76 instructions, in reality there are hundreds of variations. The 68000 uses a full 16 bit word for its operation code compared to the 6502's single byte. In addition, this word may be divided into a number of fields according to the instruction class. Bit patterns in these fields specify register numbers, the addressing mode for the operands, and in some instructions, a small ammount of "quick immediate" data. Thus, even the op code must often be "assembled" from smaller parts.

You do this assembly by simply adding together the symbolic name (which had earlier been equated to a bit pattern) for each part in an expression. For example, the 68000 instruction to add data register 1 to address register 4 might look like:

```
.DBYTE   ADDA.L+DRD+   D1+SA4
```

The .DBYTE specifies that 16 bit words are to be assembled (don't use .WORD because it assembles in reverse order). ADDA.L has been equated to the bit pattern for the "Add Address Long" 68000 instruction. DRD has been equated to the bit pattern for the "Data Register Direct" addressing mode while D1 and SA4 have been equated to the patterns for data register 1 and address register 4 (shifted to bit 9) respectively. The assembler will add these 4 parts together to form the composite bit pattern for the instruction. Note that you may have blanks after an operator in the MTU assembler. Also since things are being added together, their order makes no difference so you can put the source (D1) before the destination (SA4). These measures improve the readability of the instruction representation.

The Datamover distribution disk includes a file called OP68000.A which has all of the necessary equates for 68000 op codes, addressing modes, and register IDs. In addition, it has a brief description of the operation and operands of each instruction class. Particularly helpful are directions for assembling a complete instruction from component parts. You should look through this file with the Editor (it is also listed in Appendix 1) to get a feel for how it may be used. Also, Appendix 2 has a listing of a short 68000 program assembled using the OP68000.A equate file which should be studied as well.

The first line of the OP68000.A file defines the symbol, S, as equal to 512. This is important because in some 68000 instructions there is a data field (shift counts and quick data) that starts at bit 9 and thus the data must be multiplied by S (Shifted) before being added to the op code.

Next, the data and address register designations are equated to corresponding numerical values. By using these symbols instead of the numbers, the cross reference listing will show all references to registers. The symbols D0...D7 and A0...A6, SP are to be used when the register field is in bits 0-2 of the instruction as it usually is. A second set of register designations (SD0...SD7 and SA0...SA6, SSP) are to be used when the register field is in bits 9-11 of the instruction.

Following the register equates are the effective address mode equates. One of these must be used on all instructions that have an effective address field. For address modes that refer to a register, you would add the address mode symbol to the register designation. For example, ARII+A5 would specify Address Register Indirect post Increment addressing using address register 5.

The MOVE instruction is unique in that it has two effective address fields, one for the source and one for the destination. The source effective address field is just like any other instruction but the destination field is located elsewhere in the instruction and is backward. In order to accommodate this, there is a separate set of address mode equates specifically for the destination field of the MOVE instruction. Their names are the same but are preceeded by DM (for Destination of Move). The DM address modes may only be used in conjunction with a MOVE instruction.

The remainder of the file is filled with equates for each instruction and its major variations. The first part of the name is usually the standard 68000 mnemonic for the instruction. Following this may be qualifiers separated from the main op code by periods. Most instructions have a length qualifier such as .B for byte, .W for word, and .L for long word. There may be other qualifiers such as .EA for effective address, .R for register, .RM for register-to-memory, .ER for effective address-to-register, etc. You should refer to each instruction for the kind of qualifiers it allows. Note that the op code and qualifiers are all recognized as one symbol by the 6502 assembler thus the qualifiers must be given in the order specified by the equate. If a particular op code-qualifier combination cannot be found, then it is not available according to Motorola documentation.

In some instructions the operand data (as distinguished from an operand register) is actually part of the op code. In these cases you would simply add the operand to the op code bit pattern just like it was a register or address mode designation. Unfortunately, these operands must usually be "prepared" before they are added in. Some instructions have a 3 bit "quick" field in bits 9-11 for specifying shift counts and very small constants. You must multiply this data by S before adding it to the op code. Be sure to enclose the multiplication in brackets however since the assembler does not recognize operator heirarchy. For example, to shift data register 4 six bits left, you could write:

```
.DBYTE  LSL.L+    D4+[6*S]
```

Other instructions have a full 8 bit field for immediate data or a relative jump address. For immediate data you should prepare the operand by specifying that just the low byte should be evaluated. This will prevent negative values from messing up the op code. For example, to load -10 into data register 4, you could write:

```
.DBYTE  MOVEQ+    [<-10]+SD4
```

The brackets are necessary for the assembler to recognize the low byte operator.

For jump instructions that require an 8 bit displacement, the proper value can be computed by subtracting the location counter and adding -2 to the location you want to jump to. For example, to branch on equal to MATCH, you could write:

```
.DBYTE  BEQ+[<-*-2+   MATCH]
```

Again the brackets are necessary for recognition of the low byte operator. The -2 adjustment is necessary because the 68000's location counter has already been incremented by 2 when it computes the jump address.

Other operands take the form of an additional word or words after the op code word. These can be written on the same line along with the op code by putting a comma after the last op code symbol and then following it with the operand data or symbol. Additional operand words (up to a total of 4 in the longest possible instruction) can be appended in the same way. For example, to load $&CB8 into data register 3, you could write:

```
.DBYTE  MOVE+IMM+DMDRD+   SD3,$7CB8
```

Note that in this case it is not possible to express the source first in the operands column. As another example, to copy the memory word at LSTART to LEND, you could write:

```
.DBYTE  MOVE+ABS.W+DMABS.W,   LSTART,LEND
```

If 32 bit operands (long words) are needed, you must form them from two 16 bit halves. The MTU assembler accepts 24 bit symbol values and evaluates expressions to 24 bits which simplifies the handling of 68000 long address words. For example, if you had a data table located at TABLE3 and you wanted to load its address into address register 3, you could write:

```
.DBYTE  MOVEA.L+ABS.L+   SA3,TABLE3/65536,TABLE3
```

The expression TABLE3/65536 computes the left 16 bit part of the 32 bit operand. The right 16 bit part is formed by truncation of TABLE3 to 16 bits.

When forming relative operands for use with one of the relative addressing modes, you should keep in mind that the assembler's location counter points to the op code while the 68000's location counter points to the word being scanned. Therefore you must add -2 in evaluating the first appended word, -4 in the second, etc. For example, to multiply data register 1 by FACTOR using relative addressing, you could write:

```
.DBYTE  MULS+REL+    SD1,FACTOR-*-2
```

When actually running the MTU 6502 assembler, it will print out any error messages on the console as well as in the listing file. However if the error occurs after the first word of a multi-word instruction, the error messages listed on the console will not show the source error line. In those cases you will have to refer to the listing file to determine the exact statement in error (the line number is of little help unless you merge the OP68000.A file with your program source file).

2.6 DEBUGGING USING TRAP

The MC68000 has a number of hardware features that make debugging programs easier. One of these is its ability to recognize illegal op codes or addressing modes and generate an internal interrupt (called a Trap) that directs control to an error log routine. Because of the relatively high risk of illegal codes in hand assembled programs, it is helpful to have such an error log routine available.

The file called TRAP on the Datamover distribution disk is a very simple error log routine for debugging 68000 programs. When loaded and executed (through the Reset vector), it will set up the 68000 stack and set all of the machine exception vectors, TRAP instruction vectors, and the level 7 interrupt (NMI) to point to its log entry. It then jumps to the user's 68000 program. If an error is detected during execution, a TRAP instruction is executed, or a level 7 interrupt is generated, control is returned to TRAP which saves all of the registers and the program counter in memory and halts. You can then examine memory with regular CODOS DUMP commands to determine what went wrong.

Below is the memory map "environment" that TRAP sets up:

LOCATIONS	FUNCTION
000000-000003	Initial stack pointer, =000400
000004-000007	Entry into trap set routine, =000100
000008-00000B	Entry into user program, =000400
- -	
000010-00002F	Trap vectors for error conditions
- -	
00007C-00007F	Vector for non-maskable interrupt
000080-0000BF	Trap vectors for TRAP instructions
- -	
000100-000185	TRAP program code
- -	
000200-00021F	Eight data registers, D0-D7
000220-00023F	Eight address registers, A0-A7
000240-000243	Program counter at point of interruption
000244-000245	Complete status register
000246-000247	=0000 for TRAP, 8080 for interrupt, FFFF for error
000248-0003FF	Default stack (grows downward from 0003FF
000400-	Default entry point and user program

To use TRAP, you would first load your program into memory and then load TRAP, both with CODOS GET commands. You should arrange for your program to terminate with a TRAP instruction instead of a STOP. If the entry point to your program is $000400 and the default stack setting (000400) is acceptable, then you can start execution by releasing 68000 Reset (SET BFBE 0C). Otherwise, set the actual entry point in locations 0008:2-000B and the desired stack pointer in locations 0000:2-0003 first using SET commands.

You can determine when (or if) your program stops by looking at location 246:2. If it is $AA, the program is probably still running. It will be set to $00 if a TRAP instruction was executed or to $FF for one of the error traps. If your program seems to be in a loop, you may interrupt it by entering: SET BFBE 1C which will transfer control to the level 7 interrupt vector and stop the program with $80 in location 246. In any case, locations 200:2-245 will then contain the data registers, address registers, program counter, and status register respectively. Note that the user program is entered in supervisor state and thus can freely execute any 68000 instruction.

If the program seems to be doing nothing and cannot be interrupted (locations $200:2-27F remain $AA), then a catastrophic error may have halted the 68000. This can be determined by looking at location $BFBE. If bit 4 is not a one, then either the 68000 has halted itself or somehow the 6502 has turned this bit off. In either case, the 68000 can only be restarted be resetting (SET BFBE 0).

2.7 USING 2 DATAMOVERS IN ONE SYSTEM

Provisions have been made for installing two Datamover boards in the same MTU-130 system. With two Datamovers you may have up to three processors running simultaneously without conflict (two 68000s and a 6502) and nearly 600K (over 2M with piggyback expansion boards) of total memory in one MTU-130 system. The 6502 can access all of the memory on both Datamovers as well as its own memory. Each 68000 however can only access the memory on its own board. Successful operation of two Datamovers involves both hardware and software considerations.

Located next to U62 on the board (see section 6.2) is two pair of square metal posts. A slip-on shorting plug should be on the pair marked 1 indicating that it is set up as Board 1. It should be left there on one of your two boards and moved to the 0 position on the other board. The effect of this plug is to change the addresses of the Control, Status, and Enable registers from $BFBE-BFBF to $BFBC-BFBD.

Although moving the shorting plug resolves addressing conflicts between the control registers, both boards will still respond to bank 2 and bank 3 MTU-130 bus addresses. Generally you will want to disable Board 0 and leave Board 1 enabled unless specifically using Board 0's memory. This can be accomplished by entering: SET BFBD 08 in response to a CODOS prompt. It is best to add this command to your STARTUP.J files so that it will be done whenever the system is turned on or cold reset. You will not damage any hardware if both boards are on simultaneously but the resulting bus fight noise if they have different data and are read may make the system susceptible to unexplained crashes.

Software control of the two boards is actually quite simple. The two sets of control and status registers are easily manipulated since they have distinct addresses. To access the memory port on one of the boards, you should first disable the other board by setting bit 3 in its Control register and then clearing bit 3 in the Control register of the board to be accessed. A good convention to follow is to leave both boards disabled unless the program needs to access one of them. When your "Dual Datamover" program exits to CODOS however it should leave board 1 enabled for use by other programs that are set up for just one Datamover and might assume that it is on.

PRINCIPLES OF OPERATION

This section gives a complete circuit level description of the Datamover hardware. Understanding of this material is not required in order to successfully program the Datamover although it may be useful in resolving detail operation questions that may arise. The description is intended to be sufficiently detailed so that an engineer or experienced technician can readily understand the Datamover's operation for maintenance purposes.

Reference should also be made to sections 4-7 while studying this section. In particular the discussion is keyed to the schematic diagrams in section 7. The four schematic pages will be referred to by "schematic page numbers" which can be found in the lower right corner of the schematic pages. Conventional logic symbols are used in all cases. Signal names are referred to in all caps and inverse signals (active-low) are designated with overbars. Input signals generally enter a page from the left and output signals leave from the right. The physical schematic location of the source for input signals and all destinations of output signals is designated with 3 characters next to the signal name. The first character is the schematic page number, the second character gives the vertical location on the page, and the last character gives the horizontal location. Such "zone numbers" are also used to identify the location of circuitry being discussed. J1 is the board edge fingers that plug into the MTU-130 bus. J2 is the memory expansion plug.

3.1 6502 ADDRESS DECODING

Decoding of the 6502 addressable I/O register addresses is performed in zone D5 of schematic page 4. NAND gates U61-12 and U62 recognize when the 6502 address bus addresses one of the two I/O registers. I/O ENAB, which participates in the gating, goes high when the MTU-130's I/O address space from $BE00-BFFF is enabled and addressed. When the jumper connected to U61-13 is in the "0" position, the recognized addresses are $BFBC and $BFBD. When it is in the "1" position (which is standard), the addresses are $BFBE and $BFBF. Since BUS PHASE 2 is factored into the gating, U62-8 will go low cleanly during the latter half of the 6502 bus cycle that addresses these locations.

Below, in zone B5, U61-6 goes low during write cycles when the Control Register is addressed. Likewise, U61-8 goes low during write cycles that address the Enable Register. U38-8 goes low whenever the Status Register is addressed regardless of cycle type. The use of these decoded signals is covered in the individual register description sections.

Since the 6502 memory port on the Datamover occupies exactly 1/2 of the MTU-130's 256K address space, no memory address decoding is required. When AB17 is high, the memory port is addressed; when it is low, the port is not addressed.

3.2 ENABLE REGISTER

The Enable Register is in zone B5 of schematic page 4. It consists of U59 which is a 4 bit latch with true and complement outputs. It is loaded from 6502 data bus lines 0-3 on the trailing edge of the address decoder's output (normally $BFBF). It is cleared to zeroes by a bus reset. The least significant 3 bits of this register are the Group Selects which act like three additional address bits for the 6502 port in other circuitry. The most significant bit is a board enable which disables the 6502 memory port when it is a one.

3.3 CONTROL REGISTER

The Control Register is in zone C5 of schematic page 4. It consists of U51 which is a 4 bit latch with true and complement outputs. It is loaded from 6502 data bus lines 1-3 and 5 on the trailing edge of the address decoder's output (normally $BFBE). It is cleared to zeroes by a bus reset. Individual output bits from this register perform miscellaneous functions which are described in the appropriate sections.

3.4 STATUS REGISTER

The Status Register is not really a register at all; it is simply an 8 bit tri-state buffer that gates various internal signals onto the 6502 data bus when enabled by the address decoder (normally $BFBE). It is enabled to drive the bus only when selected by the address decoder during read cycles. The lower 4 bits (0-3) of the Status Register are just copies of the 4 bit Enable Register. Bit 4 reflects the state of the 68000's HALT signal. Bit 5 is a copy of bit 5 of the control register. Bit 6 indicates the status of the maskable interrupt to the 68000 while bit 7 indicates the status of the interrupt request from the 68000. After a reset, which will clear all of these internal signals to zeroes, reading the Status Register should give an all zeroes status.

3.5 INTERRUPT TO 68000

The Datamover provides for two interrupts to the 68000. The circuitry for these is in zone D6 of schematic page 3. Flip-flops U25-10 and U25-6 represent the level 4 and level 7 interrupt requests respectively. These flops are set by receiving a CNTL REG CLK from the address decoder (normally $BFBE) along with a one level on their respective 6502 data bus lines. Once set, they may be cleared either by a 68000 reset or by a 68000 interrupt acknowledge. Gates U20-3 and U20-11 logical-or these two possible reset sources together for each flip-flop. U25-10 also goes back to the Status Register so the 6502 can determine when the Level 4 interrupt to the 68000 is acknowledged.

The 68000 requires its interrupt requests to be encoded into a 3 bit level number which it accepts directly. When U25-6, which is the level 7 (non-maskable) interrupt, is set, it causes all 3 of the IPL lines on the 68000 to go high thus specifying level 7. When U25-4 goes high, only IPL 2 goes high which specifies level 4.

U19 is a gated 1-of-8 decoder which recognizes the 68000 interrupt acknowledge cycle and resets the request flip-flop corresponding to the level being acknowledged. An interrupt acknowledge cycle is recognized by U19 as Function Code=7 (U49-6 is low), and Address Strobe. The level being acknowledged is then decoded from 68000 address lines A1-A3 and appears on the decoder's outputs.

3.6 8MHz CLOCK GENERATOR

The 68000 microprocessor and the memory timing signal generator require a stable, accurate source of 8MHz. In addition, the 68000 requires a reasonably symmetrical square wave. Circuitry in zone A5 of schematic page 1 generates these 8MHz signals by multiplying the 1MHz BUS PHASE 2 signal from the 6502 bus by 8 with a phase locked loop. U46-6 is the voltage controlled oscillator in the phase locked loop and is just a classic Schmidt trigger R-C oscillator. The 500 ohm pot (R4) determines the oscillator's free running frequency and is set for a nominal frequency of 8.0 MHz. This simple oscillator is made to act as a voltage controlled oscillator (VCO) by connecting a resistor (2.2K, R2) to the R-C node. A voltage increase at the resistor's free end will slow the oscillator while a voltage decrease will speed up the oscillator. Although the linear VCO range is only 20% or so, this is ample for locking to the fixed crystal-controlled 1MHz frequency of the MTU-130 bus.

The 8MHz oscillator drives U50, which is an 8 bit counter. Nand gate U57-12 decodes a particular state of the lower 3 bits of this counter to produce a low-going 125NS pulse with a repetition rate of 1MHz. This is connected to U52 which acts as a phase detector for the loop. BUS PHASE 2 is the reference input to the detector. When the loop is locked, the rising edge of BUS PHASE 2 will occur midway during the pulse from U57-12. The phase detector output (U52-13 and test point 1) therefore consists of a 60NS high pulse, 60NS low pulse, and an 875NS floating period (see the timing diagram in section 4). This waveform is averaged by low-pass filter R1 and C39 to provide the control voltage for the VCO. If the oscillator should speed up, the pulse from U57-12 will come earlier which means that the phase detector's output high time will increase and its low time will decrease thus raising the average voltage across C39. This increased voltage slows down the oscillator to maintain lock. The opposite sequence would occur if the oscillator spontaneously slowed down. The exact phase of the lock between U57-12 and BUS PHASE 2 is influenced by the setting of R4.

The 8MHz waveform at U46-6 is decidedly non-symmetrical with a low time about 2.5 times greater than the high time. This non-symmetry is used to advantage in the memory cycle timing generator but is beyond the 68000's clock symmetry specs. The other Schmidt trigger in U46 along with R5 and variable capacitor C33 are used to square up the waveform. Once the oscillator has been adjusted for synchronization with the 6502 bus cycle, C33 is adjusted for a symmetrical square wave to the 68000.

3.7 MEMORY CYCLE TIMING

Memory cycle timing for the 64K dynamic RAM chips is performed by flip-flops U48-10, U56-5, and U56-9 in zone B4 of schematic page 1. Normally all three flip-flops are off when a memory cycle is not being executed. U49-8 will go high when any of the three possible memory cycle requestors (68000, 6502, or refresh) requests that a cycle begin. When such a request occurs, the first flip-flop in the chain (U48-10) will be set on the next falling edge of the non-symmetrical 8MHz clock. The putput of this flop-flop becomes the row address strobe (RAS) for the memory ICs. The next rising edge of the clock will set U56-9 about 85NS later since it is now conditioned to do so by U48-10). This becomes the column address strobe (CAS) for the memory ICs. The third flip-flop (U56-5) is set on the next falling edge of the clock and thus turns on 125NS after the beginning of the cycle. On the second falling clock edge after the cycle began (i.e., 250NS later), U48-10 is conditioned to turn back off by U56-6. U56-9 then turns off 85NS later thus completing the cycle. Another cycle may be started on the next falling clock edge thus giving 125NS between cycles and a total memory cycle time of 375NS. Note that once the cycle is started, U48-10 will ignore the cycle request input from U49-8 until the entire 375NS cycle is complete.

3.8 MEMORY CYCLE ARBITRATION

On board the Datamover are three "users" of memory cycles which are the 68000 microprocessor, the MTU-130 bus (6502 microprocessor), and the memory refresher. It is entirely possible for two or even all three of these users to be requesting a cycle at the same time. The needs of these users are quite different as well. For example, the 68000 is capable of generating almost continuous requests (one every 500NS) but is able to wait for access indefinitely. The 6502 is not quite so voracious but is unable to wait on memory access at all. Fortunately it does give several hundred nano-seconds warning before it must have its cycle. Refresh requires occasional access (every 16uS) but it can wait for awhile although not indefinitely. These different needs plus the desire to maximize the 68000's throughput in the presence of other requests rules out a simple priority or round-robbin arbitration scheme. What is actually used is a time dependent priority scheme where the priorities effectively shift during the 1.0uS 6502 bus cycle.

The cycle arbitration logic is in zones B5 and C5 of schematic page 1. For arbitration purposes, the 1.0uS 6502 bus cycle is divided into an 875NS "6502 dominant" portion and a 125NS "68000 dominant" portion. The 68000 dominant portion of the cycle occurs early in the Phase 1 portion of the cycle while the 6502 dominant portion covers the remainder of Phase 1 and all of Phase 2 (see the Monomeg CPU manual for a description of the 6502 bus cycle). These portions are distinguished by U57-8 which decodes the low 3 bits of counter U50-6 which are phase-locked to the 6502 bus cycle.

During the 6502 dominant portion of the cycle, if the memory is addressed by the 6502 address bus (6502 MEM ADRD is high), the 68000 is prevented from taking a cycle through U38-6. Refresh is unconditionally locked out through U58-8. Thus the effective priority order is: 6502 highest, 68000 lowest, refresh never. Note that the actual beginning of a 6502 cycle may be delayed by as much as 500NS by U44-6 so that it will be run when the 6502 can use it. This gap is necessary so that a memory cycle that may have started earlier can finish before the 6502 must have its cycle.

During the 68000 dominant portion of the cycle the effective priority order is: refresh highest, 68000 lowest, and 6502 never. Since a refresh request happens only occasionally, the 68000 is granted cycles most of the time. U49-12 detects the conditions necessary for allowing a 68000 cycle to start while U57-6 detects the conditions necessary for a refresh cycle. Since the 68000 dominant portion happens once per microsecond and refresh butts in only once every 16 microseconds, the 68000 is assured of greater than 900K accesses per second even if the 6502 constantly "sits" on the Datamover's memory.

Each of the three vertically stacked flip-flops (U48-6, U42-10, and U42-6) represent the granting of a memory cycle to a user. Note that all of them are connected "upsidedown" so the term "set" in this discussion means that the flop-flop is physically reset. Only one of these flip-flops may be set at a time, that is, during the memory cycle granted to the user it represents. U49-8 logical-ORs each of the cycle grants together (cycle request for the 68000) and starts the timing generator running the cycle. Each flip-flop is turned off at the end of a memory cycle by the signal at U30-3.

U42-6 is set when a cycle is granted to the 68000. It is conditioned to set by U23-6 when the 68000 wants a cycle and is allowed to have a cycle by the arbitration logic. The complex gating of U23 insures that the cycle is started at the proper time for both read and write 68000 cycles and that a "read-modify-write" 68000 cycle is interpreted as two separate cycle requests. The 68000 cycle flip-flop is allowed to be set later than the other two so that the memory cycle can start immediately after the 68000 requests it and thus not introduce any wait states (unless pre-empted of course).

U42-10 is set when a refresh cycle is granted. It is conditioned to set by U57-6 when the refresh timing counter has overflowed, there is no 68000 cycle in progress, and it is the 68000 dominent portion of the cycle. The refresh timing counter is reset through C34 and U58-10 when the refresh cycle actually starts. The counter will overflow again and set U50-8 high 16 microseconds later. As long as the refresh cycle is taken within 2uS of being requested (which it always will), no counts are lost by resetting all 4 upper bits of the counter.

U48-6 is set when a 6502 cycle is granted. It is conditioned to set by U44-6 when 6502 MEM ADRD is true and the right timing slot in the 6502 bus cycle arrives shortly after BUS PHASE 2 becomes true. The arbitration logic has already insured that no other cycle user can be starting or still running a memory cycle at this time.

3.9 REFRESH ADDRESS COUNTER

The refresh address counter is in zone A3 and C3 of schematic page 4. It consists of two 4 bit ripple counters, both inside U45. At the end of a refresh cycle, the counter is incremented by one. The fact that a ripple counter is used does not cause problems because it is incremented at the end of the refresh cycle and therefore has nearly 16uS to settle before the next cycle. Since a refresh cycle occurs every 16 microseconds, the counter will count through 128 states every 2.048 milliseconds or through 256 states every 4.096MS. Thus either 128 cycle or 256 refresh cycle memory ICs may be accommodated.

3.10 MEMORY ADDRESS MULTIPLEXOR

The memory address multiplexor must accept addresses from three different sources and combine them into 8 multiplexed lines for the memory ICs. This in effect defines 6 sources of addresses: high 68000, high 6502, high refresh, low 68000, low 6502, and low refresh. The low halves of the address words become row addresses in the RAM chips while the high halves become column addresses. Since the refresh column addresses are not significant, there need be only 5 actual inputs to the memory address multiplexor. This is accomplished with a 2-input multiplexor driving one of the inputs of a 4-input multiplexor. Note that the address multiplexor handles word addresses since the memory array is 16 bits wide.

The address multiplexor is on the right half of schematic page 4. The 2-input "pre multiplexor" is U29 and U37 which are quad 2-input multiplexor ICs. This multiplexor selects between the lower eight 68000 address bits and the output of the refresh addresss counter. The 68000 address is normally selected except during an actual refresh cycle which avoids the need for address setup time when a 68000 cycle is started.

The 4-input main multiplexor is a set of four dual 4-input multiplexor ICs (U35, 36, 27, and 28). The SEL 0 input selects between the lower and the upper half of addresses under the control of RAS RAW which is the row address clock for the memory ICs. Normal logic delays through U33-6, the multiplexor, and U26 provide adequate hold time of the row address before switching to the column address during a memory cycle. The SEL 1 input selects between 6502 address and 68000/refresh address. Again, the 68000 address is normally selected except during an actual 6502 cycle to allow the 68000 cycle to start as early as possible.

The output of the address multiplexor goes to an octal buffer (U26) which then drives the address inputs of the 32 memory chips. It also goes directly to the memory expansion connector where it would go through similar buffers to additional memory ICs. A relatively low power buffer is used to avoid generating excessive noise in the memory array while still having ample drive capability for 32 RAMs. The inversion of the 81LS96 buffer IC (which is faster and takes less power than an 81LS95) makes no difference to the memory.

The address multiplexor that has been described only handles the lower 16 word address bits from the 6502 and 68000. The higher 3 address bits needed to address 512K words (1M bytes) are handled in zone D3 of schematic page 3. Here a quad 4-input multiplexor, U43, selects between A17-A20 from the 68000 and GRP SEL 1, 2, and 4 from the 6502 (Enable Register). The fourth address bit from the 68000 allows it to address its I/O address space which is just above the 1MB memory address space. The SEL input to the multiplexor normally selects the 68000 address bits but switches to the 6502 when a 6502 cycle is granted. The output of the multiplexor goes to decoder U34 which determines which of 8 sets of 16 memory ICs is addressed (2 on-board the Datamover and up to 6 on the memory expansion connector). It also determines when the 68000 I/O address space has been selected and signals this on pin 10.

Finally, in zone B4 of schematic page 4, is circuitry that handles the least significant 6502 address bit. The memory array is organized as 16 bit words but the 6502 must address individual bytes. U40-6 takes 6502 AB0 and influenced by control reigster bit 1, generates 6502 L BYTE and 6502 R BYTE. If Control Register bit 1 is a zero, 6502 L BYTE is activated when AB0 is zero and 6502 R BYTE is activated when AB0 is a one. The selection is reversed if Control Register bit 1 is a one. This connection allows the programmer to determine whether even 6502 byte addresses refer to the left byte or the right byte of the 16 bit word (left is standard).

3.11 MEMORY ARRAY AND DRIVERS

Schematic page 2 contains the memory array and associated drivers. Logically, the memory array consists of 2 sets of 16 RAM chips each making a total of 128K 16 bit words. Physically, each IC location on the board has two memory chips stacked vertically, one on top of the other, inside a special "stacking" socket. All except 2 of the memory IC pins are connected in parallel inside the stacking socket. The row address strobe ($\overline{RAS}$, pin 4) and column address strobe ($\overline{CAS}$, pin 15) are brought down separately. In the schematic, the two sets are drawn separately with all signals except RAS and CAS connected in parallel. Note that 128K byte versions of the Datamover have just one set of RAMs installed and do not use the stacking sockets.

The Row Address Strobe ($\overline{RAS}$) of the memory ICs is driven by AND-OR-invert gates U41-6 and U47-6 which are in zone C5 of page 4. During read and write cycles for the 68000 or 6502, only the addressed set of 16 RAM chips should get the $\overline{RAS}$ strobe in order to minimize power consumption and noise. During refresh cycles however, both sets should receive $\overline{RAS}$ so that both are refreshed with one refresh cycle. This requirement is satisfied by the AND-OR-invert logic and the use of RAM ROW 0 ADRD and RAM ROW 1 ADRD signals from the upper bits memory address multiplexor and decoder.

The Column Address Strobe ($\overline{CAS}$) of the memory ICs is driven by NAND gates U55-3 and 11. $\overline{CAS}$ is driven on every non-refresh memory cycle to all RAM sets. During a read/write cycle, only the set that received $\overline{RAS}$ will actually be selected. During a refresh cycle, $\overline{CAS}$ is gated off so that the RAMs just refresh internally and do not drive their data-out lines.

In zone A6 is the logic for driving the Write Enable ($\overline{WE}$) signal to the memory ICs. Since the memory array is organized as 16 bit words but must be byte addressable, it is divided into two 8 bit halves. During read cycles, data bus gating is sufficient to select the desired byte from the pair. During write cycles to bytes, only the desired array half should receive $\overline{WE}$. The 68000 uses Upper Data Strobe (UDS) and Lower Data Strobe (LDS) to specify which byte or both for the whole word. The 6502 uses 6502 L BYTE and 6502 R BYTE to make this selection. AND-OR-invert gates U41-8 and U47-8 look at these signals and generate $\overline{WE}$ for the memory ICs at the appropriate time.

While the 68000 has a full 16 bit data bus and internally selects bytes from this bus, the 6502 has only an 8 bit data bus. Accordingly, two data bus buffers, U53 and U54, are used between the memory array and the MTU-130 backplane bus. During 6502 cycles addressing the left byte, U54 is activated to allow data to pass between the 6502 bus and MEMD8-MEMD15. When the right byte is addressed, U53 is activated which allows communication with MEMD0-MEMD7.

3.12 MEMORY EXPANSION CONNECTOR

The memory expansion connector is provided for expanding on-board memory up to 1MB by means of connecting a "piggyback" board with little more than memory ICs on it. In zones B1-D1 of page 4, connections to the 8 multiplexed RAM address lines are shown. These are intended to go through 81LS96 buffers to the additional RAM chips on the memory expansion board (1 buffer per 32 chips). Zone D1 on page 3 shows connections to the decoder which selects the addressed set of expansion RAM ICs. RAS RAW, CAS RAW, and RFSH CYCLE should be connected in a manner similar to that in zone C5 of page 2 to drive the RAS and CAS lines on the expansion board. Finally, RAM L BYTE WE and RAM R BYTE WE should go through non-inverting buffers (up to 50NS delay is acceptable) to the WE pins of the add-on memory. The 16 data lines are connected directly to the expansion connector and require no buffering on the memory expansion board. Although the memory expansion connector is not intended for interfacing peripheral circuits to the 68000, it is possible to de-multiplex the address lines provided and effect such an interface.

3.13 68000 CHIP AND INTERFACE

The MC68000 CPU chip itself is in the left portion of schematic page 3. It receives the squared up 8MHz clock directly. RESET and HALT are driven by open-collector gates connected to the control register because the 68000 is also capable of driving these lines. The interrupt request inputs, IPL0-2, were described in section 3.5. Data Transfer Acknowledge (DTACK) is simply connected to the 68000 cycle grant flip-flop. Thus the 68000 will effectively wait indefinitely for a memory cycle to be granted to it. All of the various signals for bus arbitration and bus error handling are not used (tied high).

Twenty bits of the address bus bits are used directly while the three most significant address bits are ignored. The 16 data bus bits are connected to the memory data bus through bi-directional bus buffers U17 and U21. They could not be connected directly because the 68000 may drive the bus while waiting for memory access thus interfering with a 6502 cycle that may be in progress. The bus arbitration logic built into the 68000 reacts too slowly to be of use in this case.

An ultra-high speed inverter, U18, is used to invert the UDS, LDS, AS, WE, and VMA output signals in order to minimize the delay in requesting a memory cycle and thus insure zero wait state operation under normal conditions. U49-6 recognizes a Function Code of 7 which signifies an interrupt acknowledge cycle; there is no distinction among the other function codes. Logic in zone B2 determines whether a real memory cycle, an I/O cycle, or an interrupt acknowledge cycle is taking place. In the case of a memory cycle, it generates 68K MEM CYCLE which conditions the 68000 memory cycle request logic on page 1. In the case of an I/O or interrupt acknowledge cycle (as distinguished by U34-10 going low or a Function Code of 7 respectively), it drives the VPA input of the 68000 to cause it to go through a "6800 compatible" bus cycle rather than the normal DTACK bus cycle. In the case of I/O, the 6800 compatible cycle is easier handle (DTACK is not required) while in the case of interrupt acknowledge, the 68000 will generate the vector number itself.

3.14 INTERRUPT TO 6502

Zone A2 on page 3 shows the interrupt to 6502 flip-flop and associated circuitry. AND gate U44-8 recognizes an I/O cycle from the 68000. U30-8 is activated by I/O read cycles and gates the status of U32-6 onto the least significant 68000 data bus line through open collector gate U39-6. U38-3 is activated by I/O write cycles and clocks the desired data into U32-6 from 68000 D0. Back on page 4, the output of this flop-flop is gated onto the 6502's IRQ line through U39-8. It also enters bit 7 of the Status Register. Logic in zone A6 of page 3 allows the flip-flop to be directly reset either by a 68000 reset or by the 6502 writing to the Control Register with a one in bit 7.

3.15 POWER SUPPLY

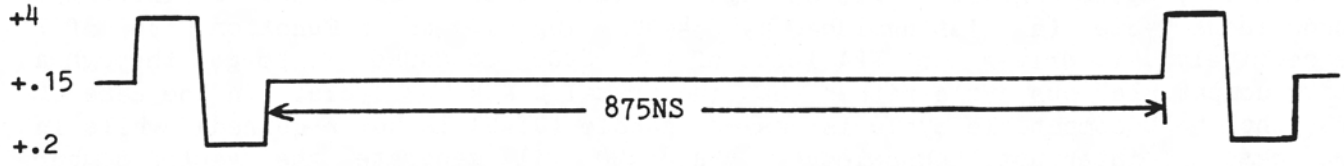
The power supply is in zone C6 of page 3. Since the only required on-board voltage is +5 volts, a simple 3-terminal 5 volt regulator IC is all that is necessary. Unregulated +8 volts input (+7V instantaneous minimum, +10 average maximum) is accepted directly from the MTU-130 bus. C1 prevents oscillations in the regulator.

4. ADJUSTMENT AND TROUBLESHOOTING

This section describes troubleshooting techniques and two adjustments on the Datamover board. It is strongly suggested that the adjustments be checked and that the suggestions given be followed prior to contacting Micro Technology Unlimited about board failures.

4.1 ADJUSTMENTS

In order for the Datamover to function properly, the phase-locked loop in the timing generator must be properly adjusted. If the Enable, Control, and Status registers seem to function normally but reading Datamover memory from the 6502 gives inconsistent results, it is possible that the factory adjustments have been disturbed or have drifted. In particular if U46 or the 5 volt regulator is replaced or the Datamover is operated from externally supplied regulated 5 volt power these adjustments should be checked. To check the PLL adjustment, obtain an oscilloscope, set it for .5uS per divison and look at the signal on test point 1. When in adjustment, this should be a clean "doublet pulse" that rests at approximately +1.5 volts for 875NS, pulses up to about +4 volts for about 60NS, immediately pulses down to +.2 volts for 60NS, and then returns to +1.5 as shown below:



The high and low portions should be equal in width and the repetition frequency should be exactly 1.0MHz. If the waveform is not as shown, remove the seal from R4 and slowly rotate it until synchronization is achieved as evidenced by a stable pattern. Further rotate R4 until the high and low parts of the waveform are equal in duration.

If readjustment of the phase-locked loop was necessary, the 68000 clock symmetry should also be checked. Using at least a 25MHz scope and 10X probe, adjust C33 until the waveform at U46-8 is symmetrical at the +1.5 volt level. Careful "eyeballing" of the waveform is adequate for either adjustment as they are not overly critical.

If the Datamover has never functioned in your system, you should first remove it, thoroughly clean the edge fingers with Scotchbrite or a soft pink pencil eraser, and then try again. If still no luck, try operating it in a different bus slot. If proper operation cannot be obtained, then the board may have been damaged in handling or there is an intermittant connection somewhere. Try reseating each of the ICs into their sockets before continuing.

If the Datamover prevents the rest of the system from functioning when it is installed, the first step is to verify that proper power is reaching the circuitry. Install the Datamover in the top card file slot so it can be reached, disconnect the speaker so the keyboard can be moved out of the way, and turn the MTU-130 power on. With a voltmeter or preferably calibrated oscilloscope, connect the ground lead to the screw holding the regulator to its heatsink. Look at the voltage on U46-14 which is just to the left of the heatsink. It should read between +4.8 and +5.2 volts and be completely free of ripple. If it reads low or has some ripple, look at the input to the regulator. This should read between about +8.5 and 9.5 volts with just the Monomeg, Disk Controller, and Datamover boards installed. The ripple frequency should be 120Hz (8.3MS between peaks). If the voltage is low (less than +8 on a voltmeter) or the ripple is 60Hz, turn the power off and check for a good connection between the power transformer secondary leads and the 10 pin white nylon connector in the MTU-130 power supply.

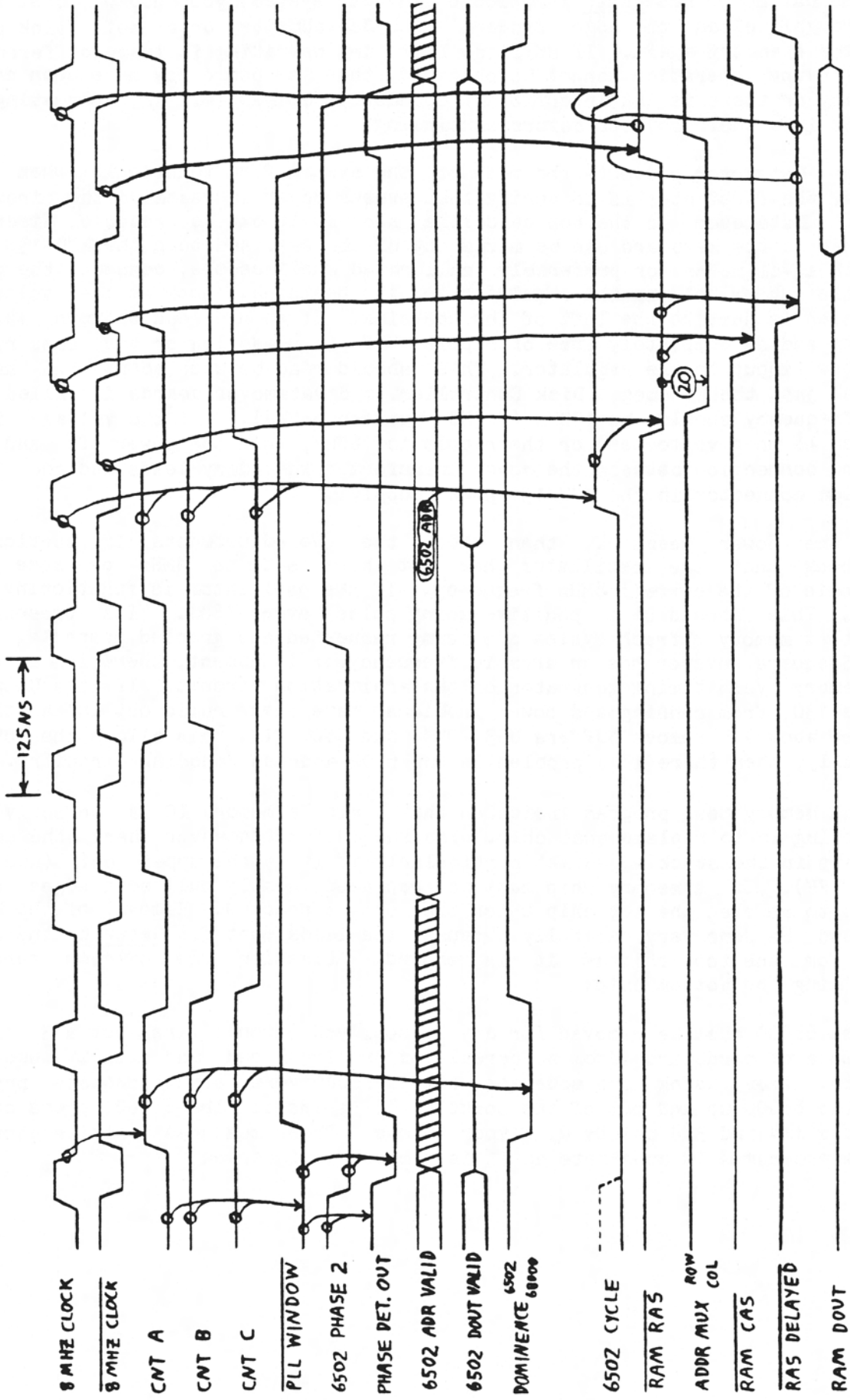
If the power seems OK, then check the two adjustments in section 4.1. Double-check that the oscillator has not been set to 4MHz or some other sub-harmonic of the correct 8MHz frequency. If the oscillator is functioning, look at U50-8. This should be a positive-going pulse every 16uS. Its presence indicates that memory refresh cycles are being requested and granted properly. If it is a 32uS square wave or has an erratic frequency or is absent, there is a problem in the memory cycle timing generator or the arbitration circuit. If the Datamover keeps the 130 from running and power problems have been ruled out as a possible cause (see above), remove buffers U53, U54, and U60. If this allows the 6502 to run normally, then there is a problem in the 6502 address decoding circuitry.

If the memory test program indicates that a single memory IC is defective, the obvious thing is to replace that chip. You should first however check the seating of the chip in the stacking socket particularly if it is the upper chip (addresses $20000-3FFFF). If a memory chip must be replaced, gently pull both edges of the socket up to release the top chip which then can be removed. Removal of the bottom chip should be done very carefully because its leads must fit between the socket contacts for the top chip as it is removed. Likewise, use extreme care when re-installing the bottom chip.

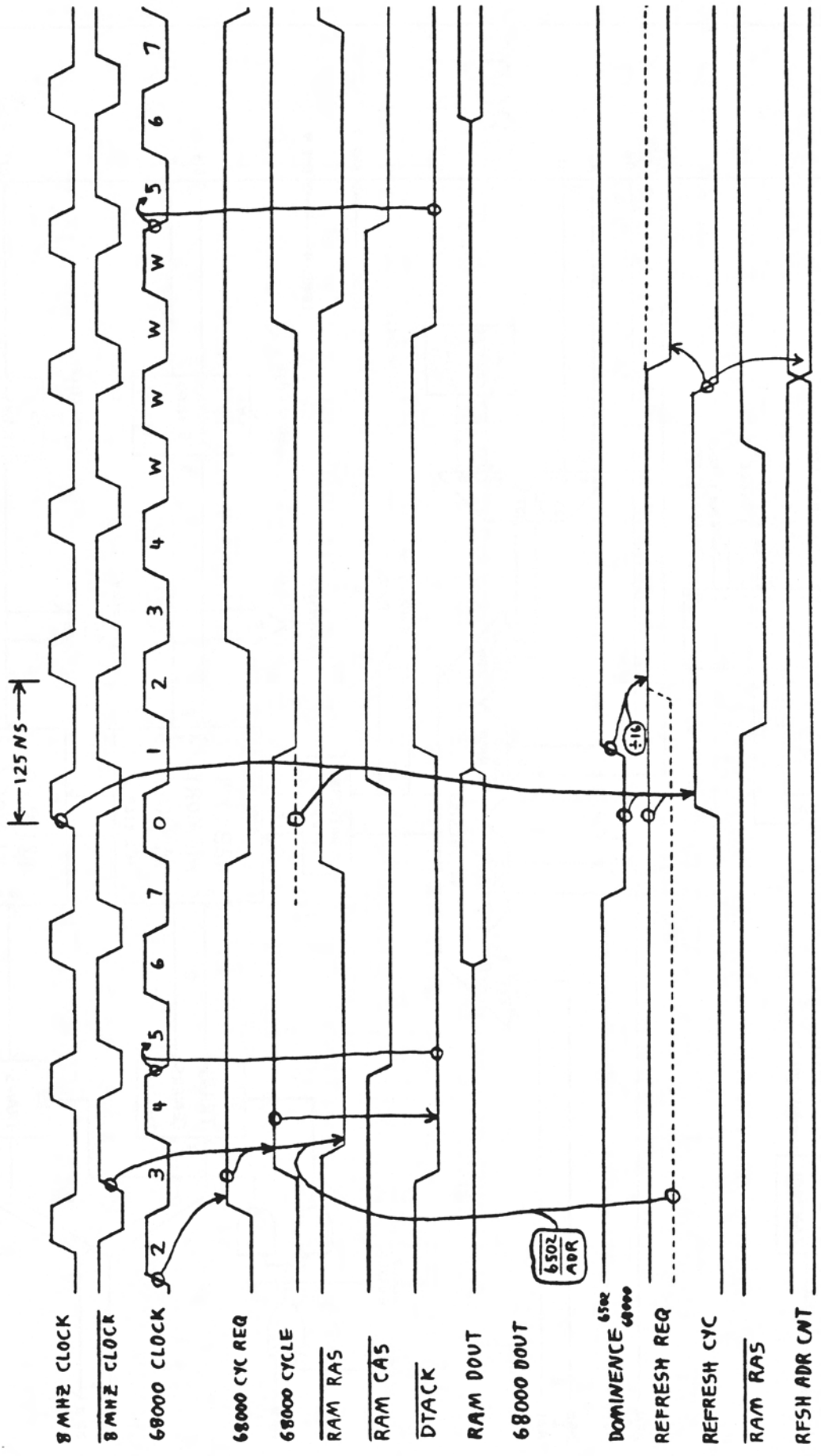
If the 68000 must be removed for any reason, you should first cut a 3 1/4" by 13/16" piece of cardboard from a cereal box or legal pad and slip it under the 68000 chip. Then, using two moderate size screwdrivers, simultaneously pry both ends of the 68000 up and out of the socket. To re-install the 68000, press on both ends at the 1/4 and 3/4 points with your thumbs. These measures minimize stress on the fragile ceramic IC substrate as it is removed or inserted.

5.

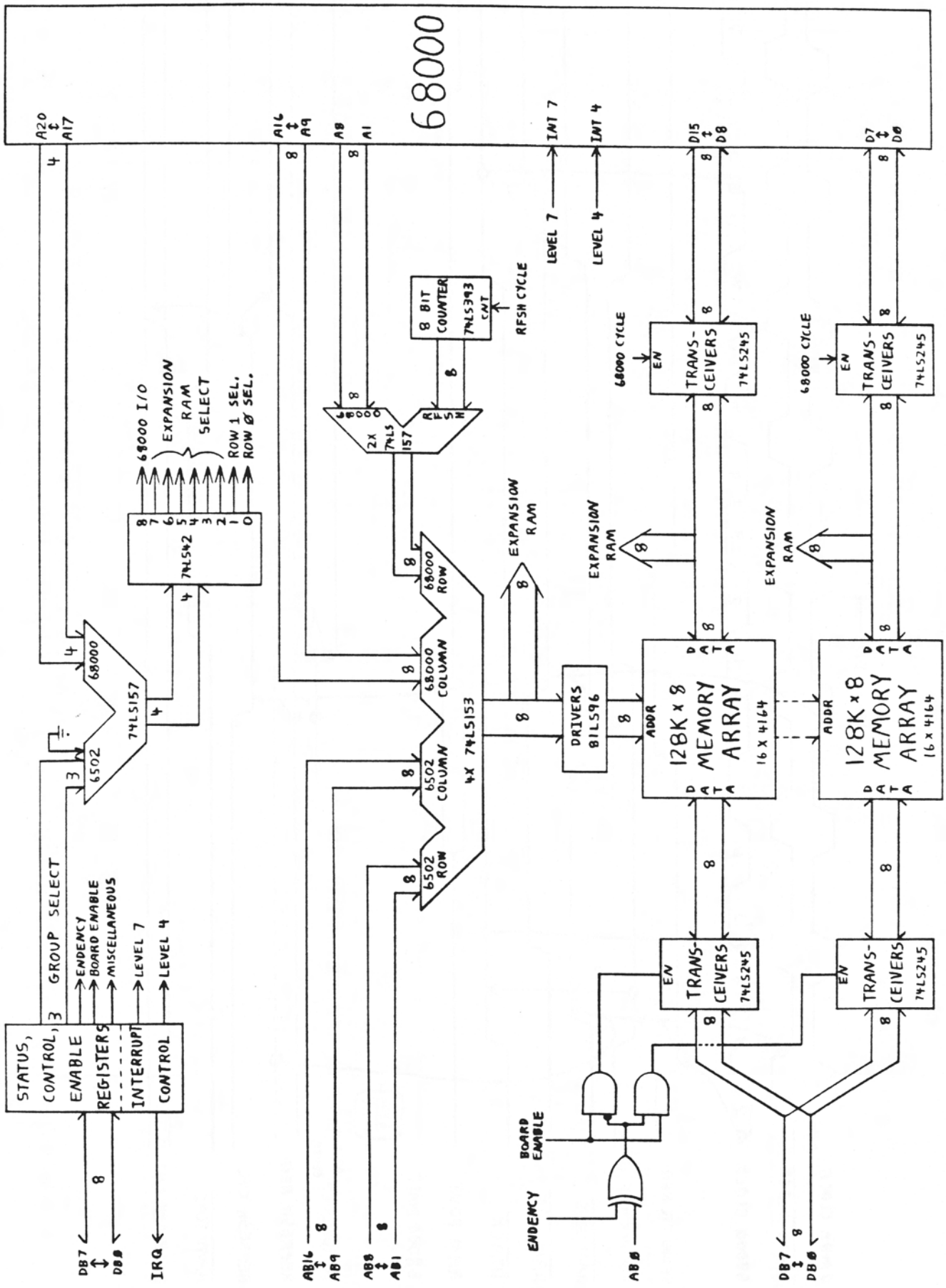
DATAMOVER TIMING 6502 CYCLES



DATAMOVER TIMING 68000 AND REFRESH CYCLES



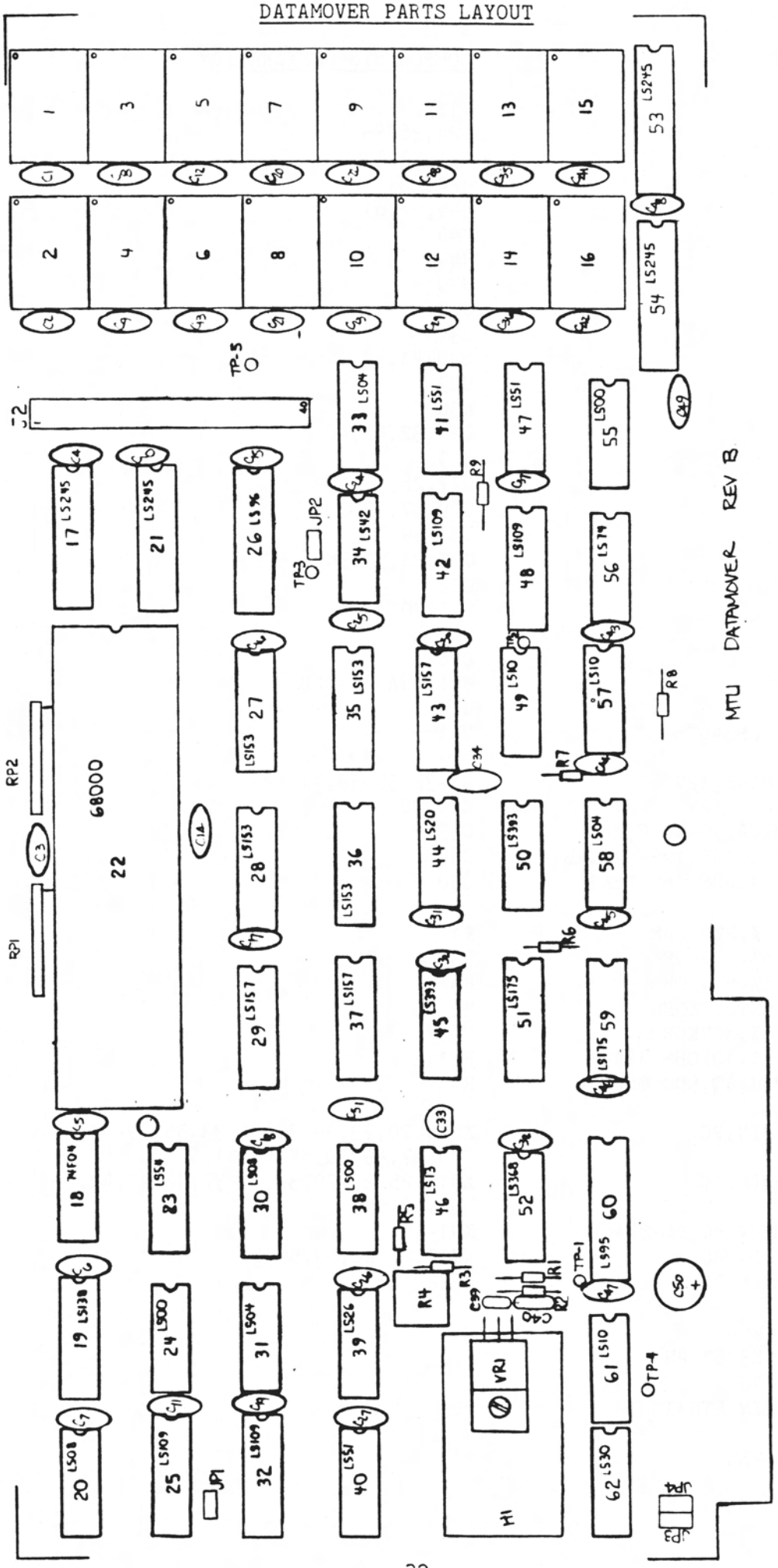
DATAMOVER BLOCK DIAGRAM



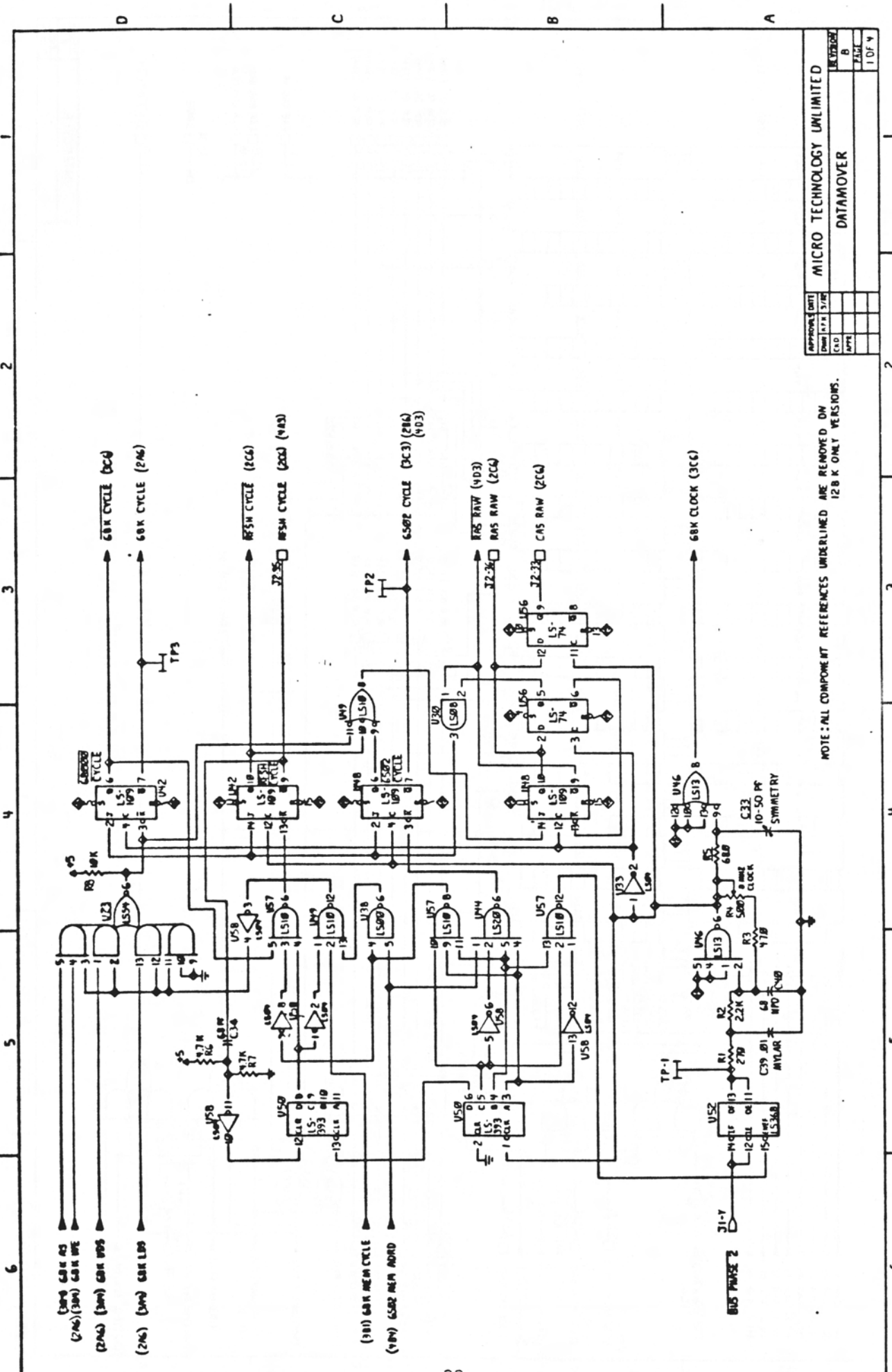
DATAMOVER PARTS LIST

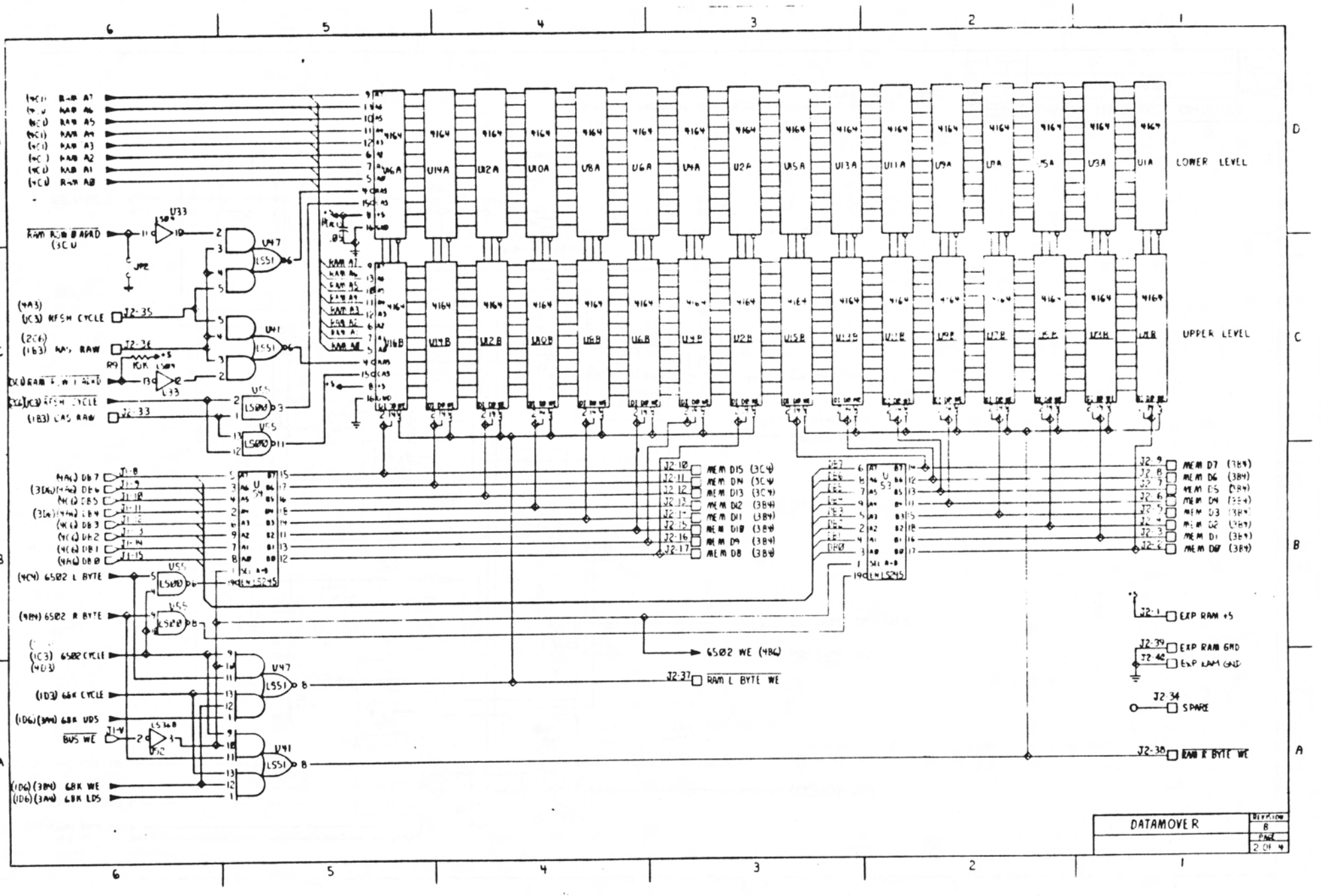
QTY	COMPONENT	COMPONENTS DESIGNATION
1	74F04	U18
3	74LS00	U24,38,55
3	74LS04	U31,33,58
2	74LS08	U20,30
3	74LS10	U49,57,61
1	74LS13	U46
1	74LS20	U44
1	74LS26	U39
1	74LS30	U62
1	74LS42	U34
3	74LS51	U40,41,47
1	74LS54	U23
1	74LS74	U56
4	74LS109	U25,32,42,48
1	74LS138	U19
4	74LS153	U27,28,35,36
3	74LS157	U29,37,43
2	74LS175	U51,59
4	74LS245	U17,21,53,54
1	74LS368	U52
2	74LS393	U45,50
1	81LS95	U60
1	81LS96	U26
32	4164	U1A-16A,1B-16B
1	MC68000	U22
1	VOLT REG, LM340T5	VR1
45	CAP,Z5U,.05UF,12V	C1-32,35-39,41-49
2	CAP,NPO,68PF,12V	C34,40
1	CAP,VAR,MICA,10-50PF,RND	C33
1	CAP,MYLAR,.01UF,100V,AXIAL	C39
1	CAP,ELECT,100UF,16V,VERT	C50
1	RES,1/4W,5%,270 OHM	R1
1	RES,1/4W,5%,470 OHM	R3
1	RES,1/4W,5%,680 OHM	R5
1	RES,1/4W,5%,2.2KOHM	R2
2	RES,1/4W,5%,4.7KOHM	R6,7
2	RES,1/4W,5%,10KOHM,RPACK	RP1,2
1	RES,TRIMPOT,SQ,500 OHM	R4
23	SOCKET,14PIN,PC	XU18,20,23,24,30,31,33,38,39-41, 44-47,49,50,55-58,61,62
16	SOCKET,16PIN,PC	XU19,25,27-29,32,34-37,42,43,48, 51,52,59
16	SOCKET,16PIN,PC,RAMSTACK	XU1-16
6	SOCKET,20PIN,PC	XU17,21,26,53,54,60
1	SOCKET,64PIN,PC	XU22
1	HEATSINK,4W	H1
2	SCREW,4-40X3/8",PH	
2	NUT,HEX,4-40	
1	HEADER,40PIN STRAIGHT	J2
1	PCB,DATAMOVER	

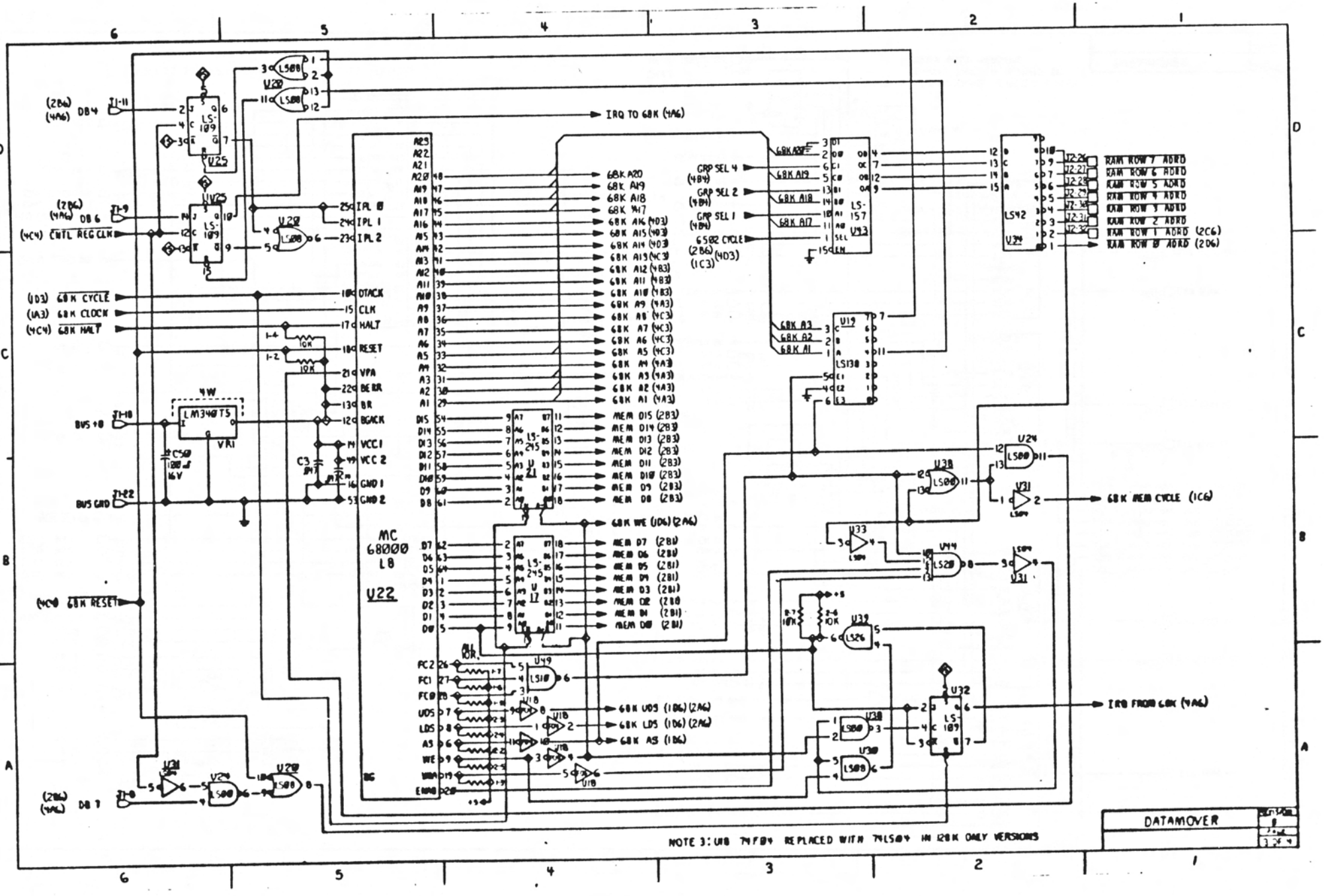
DATAMOVER PARTS LAYOUT

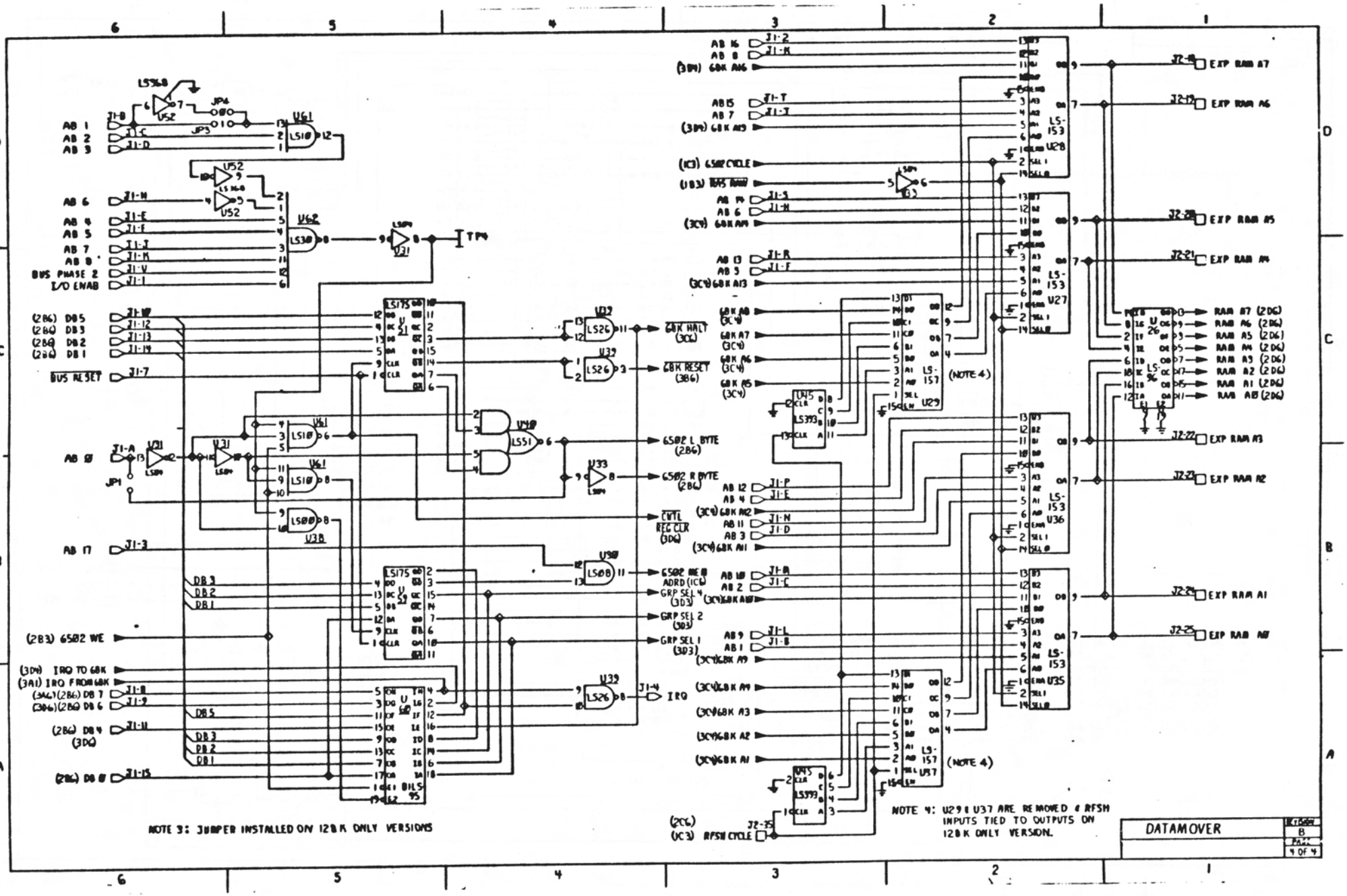


DATAMOVER SCHEMATIC DIAGRAMS




LOWER LEVEL
UPPER LEVEL
DATAMOVER
REVISION B
PAGE 2 OF 4


IRQ TO 68K (4A6)
MC 68000 L8
U22
68K RESET
68K CYCLE
68K CLOCK
68K HALT
CNTL REG CLK
DB 6
DB 7
BUS +8
BUS GND
LM340T5
VR1
IPL 0
IPL 1
IPL 2
DTACK
CLK
HALT
RESET
VPA
BERR
BR
BGACK
VCC 1
VCC 2
GND 1
GND 2
68K A20
68K A19
68K A18
68K A17
68K A16 (4D3)
68K A15 (4D3)
68K A14 (4D3)
68K A13 (4C3)
68K A12 (4B3)
68K A11 (4B3)
68K A10 (4B3)
68K A9 (4A3)
68K A8 (4C3)
68K A7 (4C3)
68K A6 (4C3)
68K A5 (4C3)
68K A4 (4A3)
68K A3 (4A3)
68K A2 (4A3)
68K A1 (4A3)
MEM DI5 (2B3)
MEM DI4 (2B3)
MEM DI3 (2B3)
MEM DI2 (2B3)
MEM DI1 (2B3)
MEM DI0 (2B3)
MEM D9 (2B3)
MEM D8 (2B3)
68K WE (1D6) (2A6)
MEM D7 (2B1)
MEM D6 (2B1)
MEM D5 (2B1)
MEM D4 (2B1)
MEM D3 (2B1)
MEM D2 (2B1)
MEM D1 (2B1)
MEM D0 (2B1)
68K UDS (1B6) (2A6)
68K LDS (1D6) (2A6)
68K AS (1B6)
GRP SEL 4 (4B4)
GRP SEL 2 (4B4)
GRP SEL 1 (4B4)
6502 CYCLE (2B6) (1C3)
68K A20
68K A19
68K A18
68K A17
68K A3
68K A2
68K A1
RAM ROW 7 ADRD
RAM ROW 6 ADRD
RAM ROW 5 ADRD
RAM ROW 4 ADRD
RAM ROW 3 ADRD
RAM ROW 2 ADRD
RAM ROW 1 ADRD (2C6)
RAM ROW 0 ADRD (2D6)
68K MEM CYCLE (1C6)
IRQ FROM 68K (4A6)
NOTE 3: U18 74F04 REPLACED WITH 74LS04 IN 128K ONLY VERSIONS
DATAMOVER


AB 1
AB 2
AB 3
AB 6
AB 4
AB 5
AB 7
AB 8
BUS PHASE 2 I/O ENAB
(2B6) DB5
(2B4) DB3
(2B2) DB2
(2B1) DB1
BUS RESET
AB 10
AB 17
(2B3) 6502 WE
(3D4) IRQ TO 68K
(3A1) IRQ FROM 68K
(3A6)(2B6) DB 7
(3B6)(2B2) DB 6
(2B4) DB4 (3D2)
(2B6) DB 0
DB 5
DB 3
DB 2
DB 1
JP4
JP5
JP1
TP9
68K HALT (3C)
68K RESET (3B6)
6502 L BYTE (2B6)
6502 R BYTE (2B4)
CNTL REG CLR (3D4)
6502 ADRD (1C)
GRP SEL 4 (3D3)
GRP SEL 2 (3D3)
GRP SEL 1 (3D3)
IRQ
AB 16
AB 8
(3B4) 68K A16
AB 15
AB 7
(3B4) 68K A15
(1C3) 6502 CYCLE
(1B3) 1MS RAW
AB 14
AB 6
(3C4) 68K A14
AB 13
AB 5
(3C4) 68K A13
68K A8 (3C4)
68K A7 (3C4)
68K A6 (3C4)
68K A5 (3C4)
AB 12
AB 4
(3C4) 68K A12
AB 11
AB 3
(3C4) 68K A11
AB 10
AB 2
(3C4) 68K A10
AB 9
AB 1
(3C4) 68K A9
(3C4) 68K A9
(3C4) 68K A8
(3C4) 68K A7
(3C4) 68K A6
(3C4) 68K A5
(2C6) (1C3) RFSH CYCLE
EXP RAM A7
EXP RAM A6
EXP RAM A5
EXP RAM A4
EXP RAM A3
EXP RAM A2
EXP RAM A1
EXP RAM A0
RAM A7 (2D6)
RAM A6 (2D6)
RAM A5 (2D6)
RAM A4 (2D6)
RAM A3 (2D6)
RAM A2 (2D6)
RAM A1 (2D6)
RAM A0 (2D6)
(NOTE 4)
(NOTE 4)
NOTE 3: JUMPER INSTALLED ON 128K ONLY VERSIONS
NOTE 4: U29 & U37 ARE REMOVED & RFSH INPUTS TIED TO OUTPUTS ON 128K ONLY VERSION.
DATAMOVER
B
4 OF 4

APPENDIX 1

Listing of OP68000.a File

```
          .PAGE  'EQUATES FOR MC68000 OP CODES AND MODIFIERS'

S         =      512              ; SHIFT FACTOR FOR MANY REGISTER SPECS AND SOME
                                  ; QUICK DATA.

; UNSHIFTED REGISTER EQUATES, USE BY ADDING TO THE OP CODE WHEN THE REGISTER
; FIELD IS IN BITS 0-2 OF THE OP CODE.

D0        =      0                ; DATA REGISTER 0
D1        =      1                ; DATA REGISTER 1
D2        =      2                ; DATA REGISTER 2
D3        =      3                ; DATA REGISTER 3
D4        =      4                ; DATA REGISTER 4
D5        =      5                ; DATA REGISTER 5
D6        =      6                ; DATA REGISTER 6
D7        =      7                ; DATA REGISTER 7
A0        =      0                ; ADDRESS REGISTER 0
A1        =      1                ; ADDRESS REGISTER 1
A2        =      2                ; ADDRESS REGISTER 2
A3        =      3                ; ADDRESS REGISTER 3
A4        =      4                ; ADDRESS REGISTER 4
A5        =      5                ; ADDRESS REGISTER 5
A6        =      6                ; ADDRESS REGISTER 6
SP        =      7                ; ADDRESS REGISTER 7 (STACK POINTER)

; SHIFTED REGISTER EQUATES, USE BY ADDING TO THE OP CODE WHEN THE REGISTER
; FIELD IS IN BITS 9-11 OF THE OP CODE.

SD0       =      D0*S
SD1       =      D1*S
SD2       =      D2*S
SD3       =      D3*S
SD4       =      D4*S
SD5       =      D5*S
SD6       =      D6*S
SD7       =      D7*S
SA0       =      A0*S
SA1       =      A1*S
SA2       =      A2*S
SA3       =      A3*S
SA4       =      A4*S
SA5       =      A5*S
SA6       =      A6*S
SSP       =      SP*S
```

```
; EQUATES FOR ALL EFFECTIVE ADDRESS MODES EXCEPT DESTINATION OF MOVE.
; USE BY ADDING TO THE OP CODE WHEN INSTRUCTION HAS AN EFFECTIVE ADDRESS FIELD.

DRD     = %000000   ; DATA REGISTER DIRECT, ADD: <reg>

ARD     = %001000   ; ADDRESS REGISTER DIRECT, ADD: <reg>

ARI     = %010000   ; ADDRESS REGISTER INDIRECT, ADD: <reg>

ARII    = %011000   ; ADDRESS REGISTER INDIRECT THEN INCREMENT, ADD: <reg>

DARI    = %100000   ; DECREMENT THEN ADDRESS REGISTER INDIRECT, ADD: <reg>

ARIO    = %101000   ; ADDRESS REGISTER INDIRECT WITH OFFSET, ADD: <reg>
                    ; FOLLOW OP CODE WITH: ,<offset>

ARIOX   = %110000   ; INDEXED ADDRESS REGISTER INDIRECT WITH OFFSET, ADD: <reg>
                    ; FOLLOW OP CODE WITH: ,[<index reg>*4096]+<offset>

ABS.W   = %111000   ; ABSOLUTE ADDRESS SHORT, FOLLOW OP CODE WITH: ,<address>

ABS.L   = %111001   ; ABSOLUTE ADDRESS LONG, FOLLOW OP CODE WITH:
                    ;                         ,<upper address>,<lower address>

REL     = %111010   ; RELATIVE, FOLLOW OP CODE WITH:  ,<address-*-2>

RELX    = %111011   ; RELATIVE INDEXED, FOLLOW OP CODE WITH:
                    ;                                ,<index reg>*256+<offset>

IMM     = %111100   ; IMMEDIATE, FOR .B OR .L, FOLLOW OP CODE WITH:  ,<data>
                    ; FOR .L, FOLLOW OP CODE WITH:  ,<upper data>,<lower data>

SR      = %111100   ; STATUS REGISTER, LEGAL FOR ONLY A FEW INSTRUCTIONS


; EQUATES FOR DESTINATION ADDRESS MODE OF MOVE.
; USE BY ADDING TO THE MOVE OP CODE ONLY.

DMDRD   = %000000*64; DATA REGISTER DIRECT, ADD: 5<reg>

; (USE MOVEA FOR ADDRESS REGISTER DIRECT)

DMARI   = %000010*64; ADDRESS REGISTER INDIRECT, ADD: 5<reg>

DMARII  = %000011*64; ADDRESS REGISTER INDIRECT THEN INCREMENT, ADD: 5<reg>

DMDARI  = %000100*64; DECREMENT THEN ADDRESS REGISTER INDIRECT, ADD: 5<reg>

DMARIO  = %000101*64; ADDRESS REGISTER INDIRECT WITH OFFSET, ADD: 5<reg>
                    ; FOLLOW OP CODE WITH: ,<offset>

DMARIOX = %000110*64; INDEXED ADDRESS REGISTER INDIRECT WITH OFFSET, ADD: 5<reg>
                    ; FOLLOW OP CODE WITH: ,[<index reg>*4096]+<offset>

DMABS.W = %000111*64; ABSOLUTE ADDRESS SHORT, FOLLOW OP CODE WITH: ,<address>

DMABS.L = %001111*64; ABSOLUTE ADDRESS LONG, FOLLOW OP CODE WITH:
                    ;                         ,<upper address>,<lower address>
```

```
; OP CODES AND COMMON VARIATIONS

ABCD.RR  = %1100000100000000 ; ADD DECIMAL DATA REGISTER TO DATA REGISTER W/XTND
                             ; ADD:  <source reg>+5<dest reg>  TO OP CODE

ABCD.MM  = %1100000100001000 ; ADD DECIMAL -(ADDR REG) TO -(ADDR REG) W/XTND
                             ; ADD:  <source reg>+5<dest reg>  TO OP CODE

ADD.ER.B = %1101000000000000 ; ADD (EFFECTIVE ADDRESS) TO DATA REGISTER
ADD.ER.W = %1101000001000000 ; ADD:  5<reg>  TO OP CODE
ADD.ER.L = %1101000010000000

ADD.RE.B = %1101000100000000 ; ADD DATA REGISTER TO (EFFECTIVE ADDRESS)
ADD.RE.W = %1101000101000000 ; ADD:  5<reg>  TO OP CODE
ADD.RE.L = %1101000110000000

ADDA.W   = %1101000011000000 ; ADD (EFFECTIVE ADDRESS) TO ADDRESS REGISTER
ADDA.L   = %1101000111000000 ; ADD:  5<reg>  TO OP CODE

ADDI.B   = %0000011000000000 ; ADD IMMEDIATE TO (EFFECTIVE ADDRESS)
ADDI.W   = %0000011001000000 ; FOR .B OR .W FOLLOW OP CODE WITH:  ,<data>
ADDI.L   = %0000011010000000 ; FOR .L FOLLOW OP CODE WITH:
                             ;                ,<upper data>,<lower data>

ADDQ.B   = %0101000000000000 ; ADD QUICK TO (EFFECTIVE ADDRESS)
ADDQ.W   = %0101000001000000 ; ADD:  [5*<data>]  TO OP CODE    0 <= data <= 7
ADDQ.L   = %0101000010000000

ADDX.RR.B= %1101000100000000 ; ADD EXTENDED DATA REG TO DATA REG
ADDX.RR.W= %1101000101000000 ; ADD:  <source reg>+5<dest reg>  TO OP CODE
ADDX.RR.L= %1101000110000000

ADDX.MM.B= %1101000100001000 ; ADD EXTENDED -(ADDR REG) TO -(ADDR REG)
ADDX.MM.W= %1101000101001000 ; ADD:  <source reg>+5<dest reg>  TO OP CODE
ADDX.MM.L= %1101000110001000

AND.ER.B = %1100000000000000 ; AND (EFFECTIVE ADDRESS) TO DATA REGISTER
AND.ER.W = %1100000001000000 ; ADD:  5<reg>  TO OP CODE
AND.ER.L = %1100000010000000

AND.RE.B = %1100000100000000 ; AND DATA REGISTER TO (EFFECTIVE ADDRESS)
AND.RE.W = %1100000101000000 ; ADD:  5<reg>  TO OP CODE
AND.RE.L = %1100000110000000

ANDI.B   = %0000001000000000 ; AND IMMEDIATE TO (EFFECTIVE ADDRESS)
ANDI.W   = %0000001001000000 ; FOR .B OR .W FOLLOW OP CODE WITH:  ,<data>
ANDI.L   = %0000001010000000 ; FOR .L FOLLOW OP CODE WITH:
                             ;                ,<upper data>,<lower data>

ASL.B    = %1110000100000000 ; ARITHMETIC SHIFT LEFT DATA REGISTER
ASL.W    = %1110000101000000 ; SHIFT COUNT IN THE INSTRUCTION WORD, 1-8
ASL.L    = %1110000110000000 ; ADD:  [<count>*5]+<shift reg>  TO OP CODE
                             ; count = 0 FOR A SHIFT OF 8

ASL.R.B  = %1110000100100000 ; ARITHMETIC SHIFT LEFT DATA REGISTER
ASL.R.W  = %1110000101100000 ; SHIFT COUNT IN ANOTHER DATA REGISTER
ASL.R.L  = %1110000110100000 ; ADD:  5<count reg>+<shift reg>  TO OP CODE

ASL.EA   = %1110000111000000 ; ARITHMETIC SHIFT (EFFECTIVE ADDRESS) LEFT 1 BIT
                             ; WORD SHIFT ONLY
```

```
ASR.B     = %1110000000000000 ; ARITHMETIC SHIFT RIGHT DATA REGISTER
ASR.W     = %1110000001000000 ; SHIFT COUNT IN THE INSTRUCTION WORD, 1-8
ASR.L     = %1110000010000000 ; ADD:  [<count>*5]+<shift reg>  TO OP CODE
                              ; count = 0 FOR A SHIFT OF 8

ASR.R.B   = %1110000000100000 ; ARITHMETIC SHIFT RIGHT DATA REGISTER
ASR.R.W   = %1110000001100000 ; SHIFT COUNT IN ANOTHER DATA REGISTER
ASR.R.L   = %1110000010100000 ; ADD:  5<count reg>+<shift reg>  TO OP CODE

ASR.EA    = %1110000011000000 ; ARITHMETIC SHIFT (EFFECTIVE ADDRESS) RIGHT 1 BIT

BCC       = %0110010000000000 ; BRANCH IF CARRY IS CLEAR
BCS       = %0110010100000000 ; BRANCH IF CARRY IS SET
BEQ       = %0110011100000000 ; BRANCH IF EQUAL
BMI       = %0110101100000000 ; BRANCH IF MINUS
BNE       = %0110011000000000 ; BRANCH IF NOT EQUAL
BPL       = %0110101000000000 ; BRANCH IF PLUS
BGE       = %0110110000000000 ; BRANCH IF GREATER OR EQUAL (SIGNED COMPARE)
BGT       = %0110111000000000 ; BRANCH IF GREATER (SIGNED COMPARE)
BLE       = %0110111100000000 ; BRANCH IF LESS OR EQUAL (SIGNED COMPARE)
BLT       = %0110110100000000 ; BRANCH IF LESS (SIGNED COMPARE)
BLS       = %0110001100000000 ; BRANCH IF LOW OR SAME (UNSIGNED COMPARE)
BHI       = %0110001000000000 ; BRANCH IF HIGH (UNSIGNED COMPARE)
BHS       = %0110010000000000 ; BRANCH IF HIGH OR SAME (UNSIGNED COMPARE)
BLO       = %0110010100000000 ; BRANCH IF LOW (UNSIGNED COMPARE)
BVC       = %0110100000000000 ; BRANCH IF OVERFLOW IS CLEAR
BVS       = %0110100100000000 ; BRANCH IF OVERFLOW IS SET
BRA       = %0110000000000000 ; BRANCH ALWAYS
BRN       = %0110000100000000 ; BRANCH NEVER (NO OPERATION)
                              ; ALL BRANCHES USE RELATIVE ADDRESSING.
                              ; FOR LESS THAN 128 BYTE DISPLACEMENT,
                              ; ADD:  [<-*-2+<address>]  TO OP CODE.
                              ; FOR GREATER THAN 128 BYTE DISPLACEMENT,
                              ; FOLLOW OP CODE WITH:  ,<address>-*-2

BCHG.R    = %0000000101000000 ; TEST A BIT IN (EFFECTIVE ADDRESS) AND CHANGE,
                              ; THE BIT NUMBER IS IN A DATA REGISTER.
                              ; ADD:  5<reg>  TO OP CODE

BCHG.I    = %0000100001000000 ; TEST A BIT IN (EFFECTIVE ADDRESS) AND CHANGE
                              ; FOLLOW OP CODE WITH:  ,<bit number>

BCLR.R    = %0000000110000000 ; TEST A BIT IN (EFFECTIVE ADDRESS) AND CLEAR,
                              ; THE BIT NUMBER IS IN A DATA REGISTER.
                              ; ADD:  5<reg>  TO OP CODE

BCLR.I    = %0000100010000000 ; TEST A BIT IN (EFFECTIVE ADDRESS) AND CLEAR
                              ; FOLLOW OP CODE WITH:  ,<bit number>

BSET.R    = %0000000111000000 ; TEST A BIT IN (EFFECTIVE ADDRESS) AND SET,
                              ; THE BIT NUMBER IS IN A DATA REGISTER.
                              ; ADD:  5<reg>  TO OP CODE

BSET.I    = %0000100011000000 ; TEST A BIT IN (EFFECTIVE ADDRESS) AND SET
                              ; FOLLOW OP CODE WITH:  ,<bit number>

BTST.R    = %0000000100000000 ; TEST A BIT IN (EFFECTIVE ADDRESS)
                              ; THE BIT NUMBER IS IN A DATA REGISTER.
                              ; ADD:  5<reg>  TO OP CODE
```

```
BTST.I   = %0000100000000000 ; TEST A BIT IN (EFFECTIVE ADDRESS)
                             ; FOLLOW OP CODE WITH:  ,<bit number>

BSR      = %0110000100000000 ; BRANCH TO SUBROUTINE, RELATIVE ADDRESS
                             ; ADD:  [<-*-2+<ADDRESS>]  TO OP CODE IF <128 AWAY
                             ; ELSE FOLLOW OP CODE WITH:  ,<address>-*-2

CHK      = %0100000110000000 ; CHECK DATA REGISTER AGAINST (EFFECTIVE ADDRESS)
                             ; AND ZERO BOUNDS.
                             ; ADD:  5<reg>  TO OP CODE

CLR.B    = %0100001000000000 ; CLEAR (EFFECTIVE ADDRESS) TO ZEROES
CLR.W    = %0100001001000000
CLR.L    = %0100001010000000

CMP.B    = %1011000000000000 ; COMPARE = SUBTRACT (EFFECTIVE ADDRESS) FROM DATA
CMP.W    = %1011000001000000 ;           REGISTER AND SET CONDITION CODES
CMP.L    = %1011000010000000 ; ADD:  5<reg>  TO OP CODE.

CMPA.W   = %1011000011000000 ; COMPARE ADDRESS = SUBTRACT (EFFECTIVE ADDRESS)
CMPA.L   = %1011000111000000 ; FROM ADDRESS REGISTER AND SET CONDITION CODES
                             ; ADD:  5<reg>  TO OP CODE.

CMPI.B   = %0000110000000000 ; COMPARE IMMEDIATE = SUBTRACT IMMEDIATE DATA FROM
CMPI.W   = %0000110001000000 ;       (EFFECTIVE ADDRESS) AND SET CONDITION CODES
CMPI.L   = %0000110010000000 ; FOR .B OR .W FOLLOW OP CODE WITH:  ,<data>
                             ; FOR .L FOLLOW OP CODE WITH:
                             ;                ,<upper data>,<lower data>

CMPM.B   = %1011000100001000 ; COMPARE MEMORY = SUBTRACT -(SOURCE ADDR REG) FROM
CMPM.W   = %1011000101001000 ;       -(DEST ADDR REG) AND SET CONDITION CODES
CMPM.L   = %1011000110001000 ; ADD:  <source reg>+5<dest reg>  TO THE OP CODE

DBCC     = %0101010011001000 ; IF CARRY IS NOT CLEAR, DECREMENT AND BRANCH
DBCS     = %0101010111001000 ; IF CARRY IS NOT SET, DECREMENT AND BRANCH
DBEQ     = %0101011111001000 ; IF NOT EQUAL, DECREMENT AND BRANCH
DBMI     = %0101101111001000 ; IF NOT MINUS, DECREMENT AND BRANCH
DBNE     = %0101011011001000 ; IF NOT NOT EQUAL, DECREMENT AND BRANCH
DBPL     = %0101101011001000 ; IF NOT PLUS, DECREMENT AND BRANCH
DBGE     = %0101110011001000 ; IF NOT GREATER OR EQUAL, DECREMENT AND BRANCH
DBGT     = %0101111011001000 ; IF NOT GREATER, DECREMENT AND BRANCH
DBLE     = %0101111111001000 ; IF NOT LESS OR EQUAL, DECREMENT AND BRANCH
DBLT     = %0101110111001000 ; IF NOT LESS, DECREMENT AND BRANCH
DBLS     = %0101001111001000 ; IF NOT LOW OR SAME, DECREMENT AND BRANCH
DBHI     = %0101001011001000 ; IF NOT HIGH, DECREMENT AND BRANCH
DBHS     = %0101010011001000 ; IF NOT HIGH OR SAME, DECREMENT AND BRANCH
DBLO     = %0101010111001000 ; IF NOT LOW, DECREMENT AND BRANCH
DBVC     = %0101100011001000 ; IF OVERFLOW IS NOT CLEAR, DECREMENT AND BRANCH
DBVS     = %0101100111001000 ; IF OVERFLOW IS NOT SET, DECREMENT AND BRANCH
DBRA     = %0101000111001000 ; DECREMENT AND BRANCH
                             ; ADD:  <reg>  TO DECREMENT TO OP CODE.
                             ; ALL BRANCHES ARE RELATIVE, FOLLOW OP CODE WITH:
                             ; ,<address>-*-2
                             ; NOTE: THIS INSTRUCTION SEEMS TO ONLY AFFECT THE
                             ;       LOW HALF OF THE DATA REGISTER!

DIVS     = %1000000111000000 ; SIGNED DIVIDE DATA REGISTER BY (EFFECTIVE ADDR)
                             ; ADD:  5<reg>  TO OP CODE

DIVU     = %1000000011000000 ; UNSIGNED DIVIDE DATA REGISTER BY (EFFECTIVE ADDR)
                             ; ADD:  5<reg>  TO OP CODE
```

```
EOR.B     = %1011000100000000 ; EXCLUSIVE-OR DATA REGISTER TO (EFFECTIVE ADDRESS)
EOR.W     = %1011000101000000 ; ADD:  S<reg>  TO OP CODE
EOR.L     = %1011000110000000

EORI.B    = %0000101000000000 ; EXCLUSIVE-OR IMMEDIATE TO (EFFECTIVE ADDRESS)
EORI.W    = %0000101001000000 ; FOR .B OR .W FOLLOW OP CODE WITH:  ,<data>
EORI.L    = %0000101010000000 ; FOR .L FOLLOW OP CODE WITH:
                              ;                ,<upper data>,<lower data>

EXG.DD    = %1100000101000000 ; EXCHANGE DATA REGISTERS
EXG.AA    = %1100000101001000 ; EXCHANGE ADDRESS REGISTERS
EXG.DA    = %1100000110001000 ; EXCHANGE DATA REGISTER WITH ADDRESS REGISTER
                              ; ADD: S<reg>+<reg>  TO OP CODE
                              ; FOR EXG.DA, PUT S BEFORE THE DATA REGISTER.

EXT.W     = %0100100010000000 ; SIGN EXTEND LOW BYTE OF DATA REGISTER TO WORD
EXT.L     = %0100100011000000 ; SIGN EXTEND LOW WORD OF DATA REGISTER TO LONG WRD
                              ; ADD:  <reg>  TO OP CODE

JMP       = %0100111011000000 ; JUMP TO EFFECTIVE ADDRESS

JSR       = %0100111010000000 ; JUMP TO SUBROUTINE AT EFFECTIVE ADDRESS

LEA       = %0100000111000000 ; LOAD EFFECTIVE ADDRESS INTO ADDRESS REGISTER
                              ; ADD:  S<reg>  TO OP CODE

LINK      = %0100111001010000 ; LINK AND ALLOCATE, ADD:  <reg>  TO OP CODE

LSL.B     = %1110000100001000 ; LOGICAL SHIFT LEFT DATA REGISTER
LSL.W     = %1110000101001000 ; SHIFT COUNT IN THE INSTRUCTION WORD, 1-8
LSL.L     = %1110000110001000 ; ADD:  [<count>*S]+<shift reg>  TO OP CODE
                              ; count = 0 FOR A SHIFT OF 8

LSL.R.B   = %1110000100101000 ; LOGICAL SHIFT LEFT DATA REGISTER
LSL.R.W   = %1110000101101000 ; SHIFT COUNT IN ANOTHER DATA REGISTER
LSL.R.L   = %1110000110101000 ; ADD:  S<count reg>+<shift reg>  TO OP CODE

LSL.EA    = %1110001111000000 ; LOGICAL SHIFT (EFFECTIVE ADDRESS) LEFT 1 BIT
                              ; WORD SHIFT ONLY

LSR.B     = %1110000000001000 ; LOGICAL SHIFT RIGHT DATA REGISTER
LSR.W     = %1110000001001000 ; SHIFT COUNT IN THE INSTRUCTION WORD, 1-8
LSR.L     = %1110000010001000 ; ADD:  [<count>*S]+<shift reg>  TO OP CODE
                              ; count = 0 FOR A SHIFT OF 8

LSR.R.B   = %1110000000101000 ; LOGICAL SHIFT RIGHT DATA REGISTER
LSR.R.W   = %1110000001101000 ; SHIFT COUNT IN ANOTHER DATA REGISTER
LSR.R.L   = %1110000010101000 ; ADD:  S<count reg>+<shift reg>  TO OP CODE

LSR.EA    = %1110001011000000 ; LOGICAL SHIFT (EFFECTIVE ADDRESS) RIGHT 1 BIT
                              ; WORD SHIFT ONLY

MOVE.B    = %0001000000000000 ; MOVE (EFFECTIVE ADDRESS) TO (MOVE EFFECTIVE ADDR)
MOVE.W    = %0011000000000000 ; ADD: DM<dest eff addr>+<src eff addr>  TO OP CODE
MOVE.L    = %0010000000000000

MOVTOCCR  = %0100010011000000 ; MOVE (EFFECTIVE ADDRESS) TO CONDITION CODES

MOVTOSR   = %0100011011000000 ; MOVE (EFFECTIVE ADDRESS) TO STATUS REGISTER

MOVFMSR   = %0100000011000000 ; MOVE STATUS REGISTER TO (EFFECTIVE ADDRESS)
```

```
MOVTOUSP = %0100111001100000 ; MOVE ADDRESS REGISTER TO USER STACK POINTER
MOVFMUSP = %0100111001101000 ; MOVE USER STACK POINTER TO ADDRESS RECISTER
                             ; ADD:  <reg>  TO OP CODE

MOVEA.W  = %0011000001000000 ; MOVE (EFFECTIVE ADDRESS) TO ADDRESS REGISTER
MOVEA.L  = %0010000001000000 ; ADD:  S<reg>  TO OP CODE

MOVEM.RM.W = %0100100010000000 ; MOVE MULTIPLE REGISTERS TO (EFFECTIVE ADDRESS)
MOVEM.RM.L = %0100100011000000 ;
MOVEM.MR.W = %0100110010000000 ; MOVE (EFFECTIVE ADDRESS) TO MULTIPLE REGISTERS
MOVEM.MR.L = %0100110011000000 ; FOLLOW INSTRUCTION WITH:  ,<mask>
                             ; BITS IN mask SPECIFY THE REGISTERS TO MOVE.

MOVEP.MR.W = %0000000100001000 ; MOVE PERIPHERAL DATA MEMORY TO REGISTER
MOVEP.MR.L = %0000000101001000 ; ADD:  S<data reg>+<address reg>  TO OP CODE
                             ; FOLLOW OP CODE WITH:  ,<displacement>

MOVEP.RM.W = %0000000110001000 ; MOVE PERIPHERAL DATA REGISTER TO MEMORY
MOVEP.RM.L = %0000000111001000 ; ADD:  S<data reg>+<address reg>  TO OP CODE
                             ; FOLLOW OP CODE WITH:  ,<displacement>

MOVEQ    = %0111000000000000 ; MOVE QUICK TO DATA REGISTER
                             ; ADD:  S<reg>+[<<immediate data>]  TO OP CODE
                             ; -128 < immediate data < 127

MULS     = %1100000111000000 ; MULTIPLY SIGNED (EFFECTIVE ADDRESS) BY DATA REG
                             ; ADD:  S<reg>  TO OP CODE

MULU     = %1100000011000000 ; MULTIPLY UNSIGNED (EFFECTIVE ADDRESS) BY DATA REG
                             ; ADD:  S<reg>  TO OP CODE

NBCD     = %0100100000000000 ; NEGATE (EFFECTIVE ADDRESS) DECIMAL WITH EXTEND

NEG.B    = %0100010000000000 ; NEGATE (EFFECTIVE ADDRESS) BINARY
NEG.W    = %0100010001000000
NEG.L    = %0100010010000000

NEGX.B   = %0100000000000000 ; NEGATE (EFFECTIVE ADDRESS) BINARY WITH EXTEND
NEGX.W   = %0100000001000000
NEGX.L   = %0100000010000000

NOP      = %0100111001110001 ; NO OPERATION

NOT.B    = %0100011000000000 ; LOGICAL COMPLEMENT (EFFECTIVE ADDRESS)
NOT.W    = %0100011001000000
NOT.L    = %0100011010000000

OR.ER.B  = %1000000000000000 ; OR (EFFECTIVE ADDRESS) TO DATA REGISTER
OR.ER.W  = %1000000001000000 ; ADD:  S<reg>  TO OP CODE
OR.ER.L  = %1000000010000000

OR.RE.B  = %1000000100000000 ; OR DATA REGISTER TO (EFFECTIVE ADDRESS)
OR.RE.W  = %1000000101000000 ; ADD:  S<reg>  TO OP CODE
OR.RE.L  = %1000000110000000

ORI.B    = %0000000000000000 ; OR IMMEDIATE TO (EFFECTIVE ADDRESS)
ORI.W    = %0000000001000000 ; FOLLOW OP CODE WITH:  ,<data>
ORI.L    = %0000000010000000
```

```
PEA      = %0100100001000000 ; PUSH EFFECTIVE ADDRESS ONTO THE STACK

RESET    = %0100111001110000 ; RESET EXTERNAL DEVICES (ON DATAMOVER THIS ONLY
                             ; RESETS PENDING INTERRUPTS TO AND FROM THE 68000)

ROL.B    = %1110000100011000 ; ROTATE LEFT DATA REGISTER WITHOUT EXTEND
ROL.W    = %1110000101011000 ; SHIFT COUNT IN THE INSTRUCTION WORD, 1-8
ROL.L    = %1110000110011000 ; ADD:  [<count>*5]+<shift reg>  TO OP CODE
                             ; count = 0 FOR A SHIFT OF 8

ROL.R.B  = %1110000100111000 ; ROTATE LEFT DATA REGISTER WITHOUT EXTEND
ROL.R.W  = %1110000101111000 ; SHIFT COUNT IN ANOTHER DATA REGISTER
ROL.R.L  = %1110000110111000 ; ADD:  5<count reg>+<shift reg>  TO OP CODE

ROL.EA   = %1110011111000000 ; ROTATE (EFFECTIVE ADDRESS) LEFT 1 BIT W/O EXTEND
                             ; WORD ROTATE ONLY

ROR.B    = %1110000000011000 ; ROTATE RIGHT DATA REGISTER WITHOUT EXTEND
ROR.W    = %1110000001011000 ; SHIFT COUNT IN THE INSTRUCTION WORD, 1-8
ROR.L    = %1110000010011000 ; ADD: [<count>*5]+<shift reg>  TO OP CODE
                             ; count = 0 FOR A SHIFT OF 8

ROR.R.B  = %1110000000111000 ; ROTATE RIGHT DATA REGISTER WITHOUT EXTEND
ROR.R.W  = %1110000001111000 ; SHIFT COUNT IN ANOTHER DATA REGISTER
ROR.R.L  = %1110000010111000 ; ADD:  5<count reg>+<shift reg>  TO OP CODE

ROR.EA   = %1110011011000000 ; ROTATE (EFFECTIVE ADDRESS) RIGHT 1 BIT W/O EXTEND
                             ; WORD ROTATE ONLY

ROXL.B   = %1110000100010000 ; ROTATE LEFT DATA REGISTER WITH EXTEND
ROXL.W   = %1110000101010000 ; SHIFT COUNT IN THE INSTRUCTION WORD, 1-8
ROXL.L   = %1110000110010000 ; ADD:  [<count>*5]+<shift reg>  TO OP CODE
                             ; count = 0 FOR A SHIFT OF 8

ROXL.R.B = %1110000100110000 ; ROTATE LEFT DATA REGISTER WITH EXTEND
ROXL.R.W = %1110000101110000 ; SHIFT COUNT IN ANOTHER DATA REGISTER
ROXL.R.L = %1110000110110000 ; ADD:  5<count reg>+<shift reg>  TO OP CODE

ROXL.EA  = %1110010111000000 ; ROTATE (EFFECTIVE ADDRESS) LEFT 1 BIT WITH EXTEND
                             ; WORD ROTATE ONLY

ROXR.B   = %1110000000010000 ; ROTATE RIGHT DATA REGISTER WITH EXTEND
ROXR.W   = %1110000001010000 ; SHIFT COUNT IN THE INSTRUCTION WORD, 1-8
ROXR.L   = %1110000010010000 ; ADD:  [<count>*5]+<shift reg>  TO OP CODE
                             ; count = 0 FOR A SHIFT OF 8

ROXR.R.B = %1110000000110000 ; ROTATE RIGHT DATA REGISTER WITH EXTEND
ROXR.R.W = %1110000001110000 ; SHIFT COUNT IN ANOTHER DATA REGISTER
ROXR.R.L = %1110000010110000 ; ADD:  5<count reg>+<shift reg>  TO OP CODE

ROXR.EA  = %1110010011000000 ; ROTATE (EFFECTIVE ADDRESS) RIGHT 1 BIT WITH EXTND
                             ; WORD ROTATE ONLY

RTE      = %0100111001110011 ; RETURN FROM EXCEPTION (RETURN FROM INTERRUPT)

RTR      = %0100111001110111 ; RETURN AND RESTORE CONDITION CODES

RTS      = %0100111001110101 ; RETURN FROM SUBROUTINE
```

```
SBCD.RR   = %1000000100000000 ; SUBTRACT DECIMAL DATA REG. TO DATA REG. WITH XTND
                              ; ADD:  <source reg>+5<dest reg>  TO OP CODE

SBCD.MM   = %1000000100001000 ; SUBTRACT DECIMAL -(ADDR REG) TO -(ADDR REG) W XTD
                              ; ADD:  <source reg>+5<dest reg>  TO OP CODE

SCC       = %0101010011000000 ; SET (EFFECTIVE ADDRESS) TO 1S IF CARRY IS CLEAR
SCS       = %0101010111000000 ; SET (EFFECTIVE ADDRESS) TO 1S IF CARRY IS SET
SEQ       = %0101011111000000 ; SET (EFFECTIVE ADDRESS) TO 1S IF EQUAL
SMI       = %0101101111000000 ; SET (EFFECTIVE ADDRESS) TO 1S IF MINUS
SNE       = %0101011011000000 ; SET (EFFECTIVE ADDRESS) TO 1S IF NOT EQUAL
SPL       = %0101101011000000 ; SET (EFFECTIVE ADDRESS) TO 1S IF PLUS
SGE       = %0101110011000000 ; SET (EA) TO 1S IF GREATER OR = (SIGNED COMPARE)
SGT       = %0101111011000000 ; SET (EA) TO 1S IF GREATER (SIGNED COMPARE)
SLE       = %0101111111000000 ; SET (EA) TO 1S IF LESS OR EQUAL (SIGNED COMPARE)
SLT       = %0101110111000000 ; SET (EA) TO 1S IF LESS (SIGNED COMPARE)
SLS       = %0101001111000000 ; SET (EA) TO 1S IF LOW OR SAME (UNSIGNED COMPARE)
SHI       = %0101001011000000 ; SET (EA) TO 1S IF HIGH (UNSIGNED COMPARE)
SHS       = %0101010011000000 ; SET (EA) TO 1S IF HIGH OR SAME (UNSIGNED COMPARE)
SLO       = %0101010111000000 ; SET (EA) TO 1S IF LOW (UNSIGNED COMPARE)
SVC       = %0101100011000000 ; SET (EA) TO 1S IF OVERFLOW IS CLEAR
SVS       = %0101100111000000 ; SET (EA) TO 1S IF OVERFLOW IS SET
                              ; ON ABOVE INSTRUCTIONS, IF CONDITION IS FALSE, SET
                              ; (EFFECTIVE ADDRESS) TO ZEROES.

SF        = %0101000011000000 ; ALWAYS SET (EA) TO ZEROES
ST        = %0101000111000000 ; ALWAYS SET (EA) TO ONES

STOP      = %0100111001110010 ; LOAD STATUS REGISTER IMMEDIATE AND STOP
                              ; FOLLOW OP CODE WITH:  ,<data>

SUB.ER.B = %1001000000000000 ; SUBTRACT (EFFECTIVE ADDRESS) FROM DATA REGISTER
SUB.ER.W = %1001000001000000 ; ADD:  S<reg>  TO OP CODE
SUB.ER.L = %1001000010000000

SUB.RE.B = %1001000100000000 ; SUBTRACT DATA REGISTER FROM (EFFECTIVE ADDRESS)
SUB.RE.W = %1001000101000000 ; ADD:  S<reg>  TO OP CODE
SUB.RE.L = %1001000110000000

SUBA.W    = %1001000011000000 ; SUBTRACT (EFFECTIVE ADDRESS) FROM ADDR REGISTER
SUBA.L    = %1001000111000000 ; ADD:  S<reg>  TO OP CODE

SUBI.B    = %0000010000000000 ; SUBTRACT IMMEDIATE FROM (EFFECTIVE ADDRESS)
SUBI.W    = %0000010001000000 ; FOR .B OR .W FOLLOW OP CODE WITH:  ,<data>
SUBI.L    = %0000010010000000 ; FOR .L FOLLOW OP CODE WITH:
                              ;                ,<upper data>,<lower data>

SUBQ.B    = %0101000100000000 ; SUBTRACT QUICK FROM (EFFECTIVE ADDRESS)
SUBQ.W    = %0101000101000000 ; ADD:  [<data>*5]  TO OP CODE
SUBQ.L    = %0101000110000000

SUBX.RR.B= %1001000100000000 ; SUBTRACT EXTENDED  DATA REG FROM DATA REG
SUBX.RR.W= %1001000101000000 ; ADD:  <source reg>+5<dest reg>  TO OP CODE
SUBX.RR.L= %1001000110000000

SUBX.MM.B= %1001000100001000 ; SUBTRACT EXTENDED -(ADDR REG) FROM -(ADDR REG)
SUBX.MM.W= %1001000101001000 ; ADD:  <source reg>+5<dest reg>  TO OP CODE
SUBX.MM.L= %1001000110001000
```

```
SWAP    = %0100100001000000 ; SWAP REGISTER HALVES, ADD:  <reg>  TO OP CODE

TAS     = %0100101011000000 ; TEST AND SET (EFFECTIVE ADDRESS)
                            ; THE OPERATION IS NOT INDIVISIBLE ON THE DATAMOVER

TRAP    = %0100111001000000 ; TRAP, ADD:  <vector #>  TO OP CODE, RANGE 0-15

TRAPV   = %0100111001110110 ; TRAP ON OVERFLOW

TST.B   = %0100101000000000 ; TEST (EFFECTIVE ADDRESS) AND SET CONDITION CODES
TST.W   = %0100101001000000
TST.L   = %0100101010000000

UNLK    = %0100111001011000 ; UNLINK, ADD:  <reg>  TO OP CODE

        .END
```

APPENDIX 2

Listing of TRAP Error Log Routine

```
TRAP SET AND PROCESS ROUTINE                    MTU 6502 ASM 1.0   4/11/82

0546 0000                .PAGE  'TRAP SET AND PROCESS ROUTINE'
0547 0000
0548 0000          ; THIS ROUTINE WILL SET THE 68000 TRAP VECTORS FOR ILLEGAL INSTRUCTIONS AND
0549 0000          ; TRAP INSTRUCTIONS AND THEN JUMP TO THE USER PROGRAM.  IF EITHER KIND OF TRAP
0550 0000          ; OCCURRS, IT WILL THEN SAVE ALL OF THE REGISTERS AND AND HALT.
0551 0000
0552 0000          ; MEMORY MAP WHEN USING TRAP:000000-000003  INITIAL STACK POINTER, =$000400
0553 0000          ;                            000004-000007  ENTRY INTO TRAP SET, =$000100
0554 0000          ;                            000008-00000B  ENTRY INTO USER PROGRAM, =$000400
0555 0000          ;                            000010-00002F  TRAP VECTORS FOR ERROR CONDITIONS
0556 0000          ;                                  -
0557 0000          ;                            000080-0000BF  TRAP VECTORS FOR TRAP INSTRUCTIONS
0558 0000          ;                                  -
0559 0000          ;                            000100-0001FF  TRAP SET AND PROCESS ROUTINE
0560 0000          ;                            000200-00021F   EIGHT DATA REGISTERS, D0-D7
0561 0000          ;                            000220-00023F   EIGHT ADDRESS REGISTERS, A0-A7 (SP)
0561 0000          ;
0562 0000          ;                            000240-000243   PROGRAM COUNTER
0563 0000          ;                            000244-000245   COMPLETE STATUS REGISTER
0564 0000          ;                            000246-000247   0000=TRAP, 8080=INTR, FFFF=ERROR
0565 0000          ;                            000248-0003FF  DEFAULT STACK
0566 0000          ;                            000400-        USER PROGRAM, DEFAULT EP = $000400
0567 0000
0568 0000          ;       EQUATES
0569 0000
0570 0010  =     ILGVEC  =       $10          ; LOCATION OF 8 ERROR CONDITION VECTORS
0571 0080  =     TRAPVC  =       $80          ; LOCATION OF 16 TRAP INSTRUCTION VECTORS
0572 0400  =     DFLTSP  =       $400         ; DEFAULT LOCATION OF STACK (GROWS DOWN)
0573 0400  =     DFLTEP  =       $400         ; DEFAULT LOCATION OF USER PROGRAM
0574 2700  =     SVSTAT  =       $2700        ; SYSTEM STATUS REGISTER WORD USED IN HALT
0575 0000                                     ; ALLOW ONLY NMI, SUPERVISOR STATE, TRACE OFF
0576 0000
0577 0000                *=     $BFBE        ; HALT AND RESET THE 68000 BEFORE LOADING TRAP
0578 BFBE 00             .BYTE  0,0
0578 BFBF 00
0579 BFC0
0580 BFC0                .BANK  2            ; ASSEMBLE INTO DATAMOVER MEMORY
0581 BFC0
0582 BFC0                *=     $200         ; TRAP LOG OUT AREA
0583 0200          DOREG *=*+    32           ; LOG OUT AREA FOR DATA REGISTERS
0584 0220          AOREG *=*+    32           ; LOG OUT AREA FOR ADDRESS REGISTERS
0585 0240          PC    *=*+    4            ; LOG OUT AREA FOR PROGRAM COUNTER
0586 0244          SSR   *=*+    2            ; LOG OUT AREA FOR SYSTEM STATUS REGISTER
0587 0246          ILLFLG *=*+   2            ; TRAP TYPE FLAG, 8000=TRAP, FFFF=ERROR
0588 0248
0589 0248                *=0                  ; RESET VECTOR
0590 0000 0000           .DBYTE 0,DFLTSP      ; INITIAL STACK POINTER
0590 0002 0400
0591 0004 0000           .DBYTE 0,INIT        ; ENTRY INTO SET TRAPS ROUTINE
0591 0006 0100
0592 0008 0000           .DBYTE 0,DFLTEP      ; ENTRY INTO USER PROGRAM
0592 000A 0400
0593 000C
0594 000C                *=$100               ; SET TRAPS ROUTINE
0595 0100
```

```
TRAP SET AND PROCESS ROUTINE                    MTU 6502 ASM 1.0   4/11/82

0596 0100 41F8   INIT     .DBYTE LEA+ABS.W+          SA0,ILGVEC    ; GET ADDRESS TO ERROR VECTORS
0596 0102 0010
0597 0104 203C            .DBYTE MOVE.L+IMM+DMDRD+   SD0,0,PTRAPE ; GET VALUE TO STORE IN VECTORS
0597 0106 0000
0597 0108 0144
0598 010A 7207            .DBYTE MOVEQ+              7+SD1         ; NUMBER OF VECTORS TO SET
0599 010C 20C0   INITL1   .DBYTE MOVE.L+DRD+DMARII+  D0+SA0        ; STORE A VECTOR AND INCR A0
0600 010E 51C9            .DBYTE DBRA+               D1,INITL1-*-2; REPEAT 8 TIMES
0600 0110 FFFC
0601 0112
0602 0112 41F8            .DBYTE LEA+ABS.W+          SA0,TRAPVC    ; GET ADDRESS TO TRAP VECTORS
0602 0114 0080
0603 0116 203C            .DBYTE MOVE.L+IMM+DMDRD+   SD0,0,PTRAPT ; GET VALUE TO STORE IN VECTORS
0603 0118 0000
0603 011A 015E
0604 011C 720F            .DBYTE MOVEQ+              15+SD1        ; NUMBER OF VECTORS TO SET
0605 011E 20C0   INITL2   .DBYTE MOVE.L+DRD+DMARII+  D0+SA0        ; STORE A VECTOR AND INCR A0
0606 0120 51C9            .DBYTE DBRA+               D1,INITL2-*-2; REPEAT 16 TIMES
0606 0122 FFFC
0607 0124
0608 0124 21FC            .DBYTE MOVE.L+IMM+DMABS.W, 0,PTRAPI,$7C ; SET ADDRESS TO INTERRUPT VECT
0608 0126 0000
0608 0128 0178
0608 012A 007C
0609 012C
0610 012C 41F8            .DBYTE LEA+ABS.W+          SA0,DOREG     ; FILL LOGOUT AREA WITH $AA
0610 012E 0200
0611 0130 203C            .DBYTE MOVE.L+IMM+DMDRD+   SD0,$AAAA,$AAAA  ; SO CAN TELL WHEN TRAP DONE
0611 0132 AAAA
0611 0134 AAAA
0611 0136 AAAA   ;
0612 0136 721F            .DBYTE MOVEQ+              31+SD1        ; FILL 128 BYTES = 32 LONG WORDS
0612 0138 721F   ;
0613 0138 20C0   INITL3   .DBYTE MOVE.L+DRD+DMARII+  D0+SA0        ; FILL A WORD AND INCR A0
0614 013A 51C9            .DBYTE DBRA+               D1,INITL3-*-2; REPEAT 32 TIMES
0614 013C FFFC
0615 013E
0616 013E 2078            .DBYTE MOVEA.L+ABS.W+      SA0,$0008     ; JUMP TO USER PROGRAM INDIRECT
0616 0140 0008
0617 0142 4ED0            .DBYTE JMP+ARI+            A0            ; THROUGH LOCATION 8
0618 0144
0619 0144         ; ERROR CONDITION TRAP SERVICE ROUTINE
0620 0144
0621 0144 48F8   PTRAPE   .DBYTE MOVEM.RM.L+ABS.W,   $FFFF,DOREG  ; LOG D0-D7 & A0-A7
0621 0146 FFFF
0621 0148 0200
0622 014A 31D7            .DBYTE MOVE.W+ARI+DMABS.W+ SP,SSR        ; LOG THE STATUS REGISTER
0622 014C 0244
0623 014E 21EF            .DBYTE MOVE.L+ARIO+DMABS.W+SP,2,PC       ; LOG THE POINT OF INTERRUPTION
0623 0150 0002
0623 0152 0240
0624 0154 31FC            .DBYTE MOVE.W+IMM+DMABS.W, $FFFF,ILLFLG ; SET ERROR TRAP FLAG
0624 0156 FFFF
0624 0158 0246
0625 015A 4E72            .DBYTE STOP,               SVSTAT        ; STOP
```

```
0625 015C 2700
0626 015E
0627 015E          ; USER TRAP SERVICE ROUTINE
0628 015E
0629 015E 48F8   PTRAPT  .DBYTE MOVEM.RM.L+ABS.W,   $FFFF,DOREG  ; LOG D0-D7 & A0-A7
0629 0160 FFFF
0629 0162 0200
0630 0164 31D7           .DBYTE MOVE.W+ARI+DMABS.W+ SP,SSR       ; LOG THE STATUS REGISTER
0630 0166 0244
0631 0168 21EF           .DBYTE MOVE.L+ARIO+DMABS.W+SP,2,PC      ; LOG THE POINT OF INTERRUPTION
0631 016A 0002
0631 016C 0240
0632 016E 31FC           .DBYTE MOVE.W+IMM+DMABS.W, $0000,ILLFLG ; SET USER TRAP FLAG
0632 0170 0000
0632 0172 0246
0633 0174 4E72           .DBYTE STOP,               SVSTAT       ; STOP
0633 0176 2700
0634 0178
0635 0178          ; INTERRUPT TRAP SERVICE ROUTINE
0636 0178
0637 0178 48F8   PTRAPI  .DBYTE MOVEM.RM.L+ABS.W,   $FFFF,DOREG  ; LOG D0-D7 & A0-A7
0637 017A FFFF
0637 017C 0200
0638 017E 31D7           .DBYTE MOVE.W+ARI+DMABS.W+ SP,SSR       ; LOG THE STATUS REGISTER
0638 0180 0244
0639 0182 21EF           .DBYTE MOVE.L+ARIO+DMABS.W+SP,2,PC      ; LOG THE POINT OF INTERRUPTION
0639 0184 0002
0639 0186 0240
0640 0188 31FC           .DBYTE MOVE.W+IMM+DMABS.W, $8080,ILLFLG ; SET INTERRUPT TRAP FLAG
0640 018A 8080
0640 018C 0246
0641 018E 4E72           .DBYTE STOP,               SVSTAT       ; STOP
0641 0190 2700
0642 0192
0643 0192                .END
   0 ERRORS IN PASS 2

   0 ERRORS IN PASS 1
```