68000 Hardware Manual

Peter A. Stark

CONTENTS

0.	Introduction	1
1.	Overview of a Micro Computer	3
2.	The Power Supply	13
3.	LED Indicators	21
4.	The RESET Circuit	27
5.	Master Clock Circuit	31
6.	68000 Operation	35
7.	68000 Operation in an Open Loop	47
8.	The MAP Circuit	53
9.	The Bus Error Circuit	59
10.	The Address Decoder	63
11.	The DTACK Circuit	71
12.	ROM and Static RAM	75
13.	The Magic Moment: First Signs of Life	79
14.	Serial Interface	87
15.	PC-Compatible Keyboard	93
16.	PC-Compatible Bus Connectors	97
17.	Computer Memories	105
18.	DRAM Circuitry	113
19.	Floppy Disk Controller	127
20.	Parallel Printer Port	135
21.	Optional -HDO Hard Disk Port	139
22.	Loose Ends	143
	APPENDICES	
A.	SK68K Parts List	145
B.	Computer Number Systems	149
C.	How to Solder	171
D.	Disk Organization	175

Chapter 0

Introduction

The microprocessor has brought about a revolution in computing. Whereas twenty or thirty years ago, computers were so expensive that only large organizations could afford them, the microprocessor has brought computers not just into the small company, but also into the home, the small office, and even into industrial and consumer equipment.

At this time, microprocessors are broken down into two major categories: CISC and RISC.

CISC microprocessors are *Complex Instruction Set Computers*. These are the traditional processors, which can execute a relatively large number of fairly complex instructions. RISC processors, on the other hand, are *Reduced Instruction Set Computers*, which can perform a smaller number of relatively simpler instructions. Right now, there is some disagreement about which are better. RISC processors, because they do less in each instruction, can be built so they do it faster. But this is outweighed to some extent by the fact that a larger number of instructions is needed to finish a particular job, and so it is not quite so clearcut which is faster on complex jobs.

In the CISC world, there are two major companies building microprocessors: Intel and Motorola. Intel was the very first microprocessor manufacturer, and Motorola was the second, and they have been the two major players ever since. Both make very capable products, and are forever playing leapfrog, trying to best each other. At any given time, one or the other may be out in front.

Although Intel is the more widely known microprocessor manufacturer - mainly because their 8088 and its followers were chosen by IBM for their PC line - many people prefer Motorola processors. Intel has attempted to make each processor they make somewhat compatible with earlier processors, with the result that even their newest processors have many features which seem archaic by today's standards. Motorola, on the other hand, made a major departure from earlier processors when they designed the

68000 - they tried to keep the “flavor" of their earlier machines, but they did not keep any of their limitations. Thus many users agree that the Motorola 68000 is easier to use - and certainly more pleasant to use - than the comparable Intel processors.

When higher level languages are used, it doesn’t much matter to the programmer which processor they will eventually run on. But to get the utmost speed or compactness out of a program requires assembly language, and that is where programmers most often choose Motorola processors, which are much more orderly in their structure and easier to program on that level.

This book is about the 68000 Microprocessor and how it is used. We will cover many details of both its internal operation, the external circuitry it connects to, and the programs which it runs. Although Motorola makes more advanced 68000-family processors (such as the 68020 and 68030), almost all of the principles you will learn about the 68000 apply to those as well - they are merely extensions of the basic 68000 architecture.

When you look at the heavy-duty number-crunching applications of microprocessors, you will find the Motorola 68000 family used more often than any other. Even though the 68000 may be one of the slower microprocessors in the series, it is no slouch.

The 68000 is the microprocessor that gives the Atari ST, Commodore Amiga, and the Macintosh SE their power. It is also found inside many laser printers, as well as in industrial controllers and scientific workstations. The 68000 is roughly in the middle of what many call the ’68K’ family of processors - the 68008 is slightly below, the 68020 and 68030 are above. (A fifth processor, the 68010, is theoretically faster than the 68000, but the 68000 can be run at faster clock rates and so is just about equal in practical speed.)

Our method for teaching you about the 68000 is to have you build and use an actual system.

In Volume I, you will concentrate on hardware. You will start from the very beginning, mounting one part after another, until you have a complete system up and running. As we go, we will discuss each part of the system, see where it fits into the overall picture, build it, and then test it.

In Volume II, you will then use the hardware to learn about software. Using the HUMBUG ROM-based debugger, and the SK*DOS disk operating system, we will progressively do some simple programming exercises, and ultimately look at various portions of the HUMBUG and SK*DOS software code itself to examine how 68000 programs solve various programming problems, and how to interface the software with the hardware.

Above all, *don’t try to skip ahead in the book,* because the treatment is logically organized in a progression. If you skip some part as you go, you will find yourself missing some crucial knowledge you would need later.

With that, let us continue to Chapter 1, to learn about the SK68K computer you will be building.

Chapter 1

Overview Of A Micro Computer

The typical computer consists of the following parts:

1. The Central Processing Unit or CPU, which does the actual processing, arithmetic, and decision-making of the computer, and also controls the operation of the rest of the computer.
2. The memory, which stores programs being performed, as well as data and results.
3. Input and output equipment (also called I/O devices), which includes things like printers, keyboards, and the like. Furthermore, there are circuits called *I/O interfaces* which connect the I/O devices to the CPU and memory. (This definition leaves the status of things like disk drives and tape drives a bit hazy - some people would include these in the memory category, and some in the I/O category. We will use the latter.)

With this breakdown in mind, we can look at the types of computers. Historically, as well as in size order, computers can be broken down into four types:

a. Mainframe computers are the very large ones, which often occupy an entire room (perhaps even a very *large* room). Originally, of course, all computers were this large; these days, computers like these are used by the Internal Revenue Service, large corporations, or universities for commercial or research applications. In a typical mainframe computer, the CPU might be in one floor-standing cabinet, the memory in another, I/O interfaces in another, and the actual I/O devices in a few more.
b. Minicomputers are smaller and newer, usually occupying one or more cabinets standing on the floor, but small enough to fit into an ordinary office or laboratory, along with other equipment. It too could be used for business or research applications, but is not as powerful as the very

large mainframes. In a typical minicomputer, The CPU might be one or more printed circuit boards, the memory might be a few more, and each I/O interface might be one or more pc boards as well. But all of these might be mounted in the same cabinet.

c. Micro computers are generally quite small, usually in desk-top cabinets, and less powerful than either of the above. But, given enough memory, these too can be used for both business and research applications. In most cases, the CPU is just one or two integrated circuits (*IC*s) occupying just part of a printed circuit board, which might even contain some memory or other circuitry as well.

d. Microcomputers (notice that we are now spelling it as one word, not two) are the smallest of the lot. In a typical microcomputer, a single integrated circuit contains the CPU, some memory, and even some of the I/O interfaces as well. In fact, the entire IC might be called a microcomputer. They are almost used strictly for control applications - for example, you might find one inside a camera or inside a traffic light.

The 68000 Microprocessor (also called a Micro Processing Unit, or MPU) would be the CPU in a Micro Computer. Actually, some of the 68000's faster cousins are powerful enough that they can do jobs often reserved for minicomputers in the past. And so the distinction between the minicomputer and the micro computer is becoming blurred as time goes on. In fact, some people believe that the minicomputer may even cease to exist, as micro computers take over many of their jobs.

The SK68K Computer Trainer

As shown in the photograph of Fig. 1-1, the SK68K microcomputer trainer is all contained on one printed circuit board. It contains the following:

a. The 68000 microprocessor is the large integrated circuit near the bottom.

b. The main memory consists of the 32 small integrated circuits in the bottom left corner plus the four medium size IC's near the center. The 32 ICs in the corner provide 1 megabyte of dynamic RAM (the main random-access memory); the four IC's in the center provide 4K (4 kilobytes) of static RAM with battery back up (the two slightly smaller ICs), and 32K of ROM (read-only-memory) with space for more (in the two slightly larger ICs).

c. The I/O interfaces are located mostly along the top edge and the top right of the board, and include 4 serial ports to connect to terminals, modems, printers, etc., two parallel ports for printers or other I/O devices, a floppy disk interface for up to four drives, a sound interface for a speaker, a clock/calendar chip, an interface for a PC or XT clone keyboard, and six interface connectors for additional clone-compatible I/O boards. (The clones we are talking about are the Japanese, Taiwanese or Korean computer components often known as IBM-compatible because they are interchangeable with those of IBM PC- and XT-style computers.)

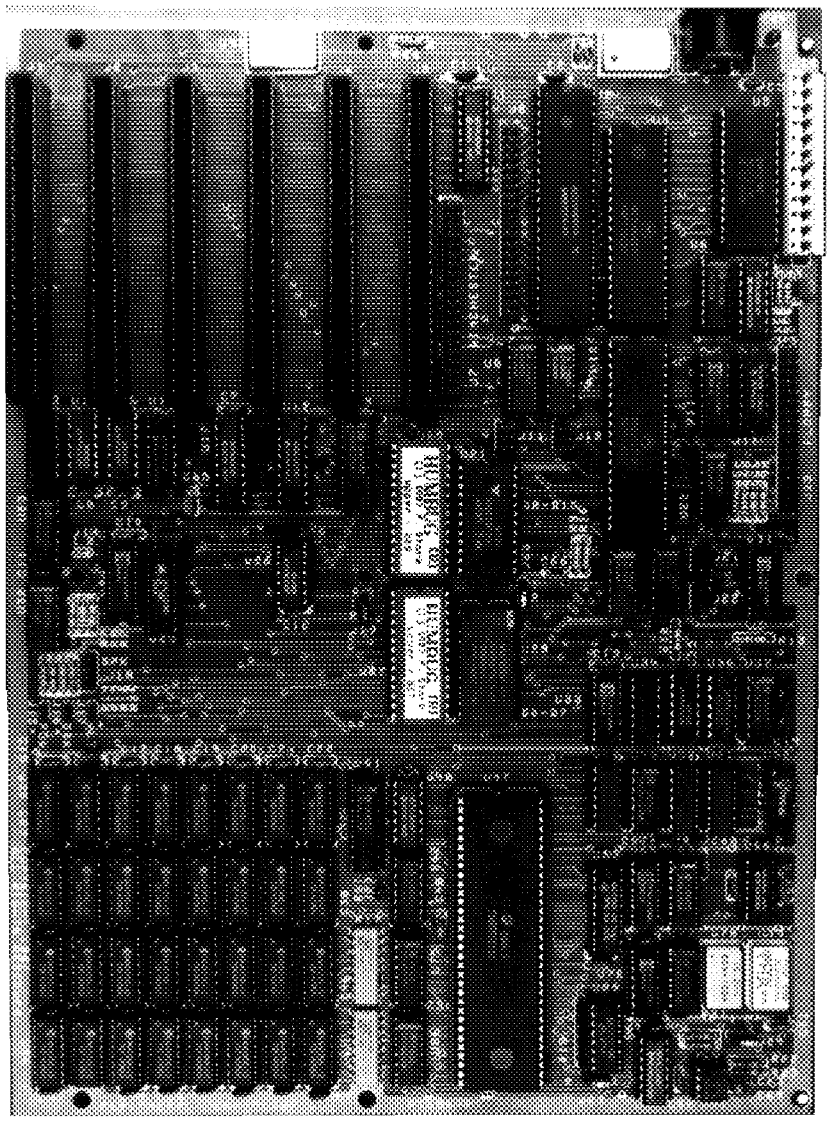


Fig. 1-1. The SK68K Printed Circuit Board.

d. Finally, the rest of the board contains various smaller ICs used for timing and control, and to interconnect the remaining parts together. In fact, these remaining ICs are often called *glue chips* for that reason.

The fact that the SK68K has connectors to accept XT-compatible clone components such as keyboards, video boards, and hard disk controllers, makes the SK68K somewhat unique. It allows us to expand the Trainer and give it additional capabilities, and do so at a very low cost by taking advantage of the fact that such clone components are produced in very large quantities and are therefore very inexpensive.

The Block Diagram

The best way to get an overall view of the SK68K trainer and how it works is to start with the block diagram in Fig. 1-2. In general terms, this diagram describes any microcomputer, not just the SK68K.

The heart of the diagram is the microprocessor, a Motorola 68000 in our case. It is driven by a clock, which is nothing more than a high frequency oscillator which generates a square wave. In the SK68K, this clock will most likely be an 8 MHz signal, though it could go as high as 16 MHz. The clock synchronizes everything occurring in the system so that it occurs at a fixed speed.

The right side of the diagram contains three essential parts: ROM, RAM, and I/O interfaces. ROM and RAM are both part of the computer's memory, but ROM can only be read (hence, its name read-only-memory or ROM), meaning that the data and programs in the ROM were placed there once the factory and can now be used by the trainer but not changed. RAM, on the other hand, is read-write- memory (somewhat misnamed as RAM rather than RWM - but then, have you ever tried to *pronounce* RWM?) The Trainer can write (store) data or programs in RAM, can later read them back, and can also change them at any time. Furthermore, ROM is permanent even when the power is turned off, whereas RAM is erased when power disappears (unless special precautions are taken, such as providing a small battery to keep the RAM powered up even when the rest of the system is shut off. Such memory is called *battery backed-up*.)

The ROM

The ROM in the SK68K system consists of two 28-pin ICs called EPROMs or Erasable Programmable ROMs. If you purchased these from an electronic distributor, they would be empty or erased. But the two EPROMs which come with the SK68K kit have been programmed with a copy of the HUMBUG debugging program and with a Basic translator. The

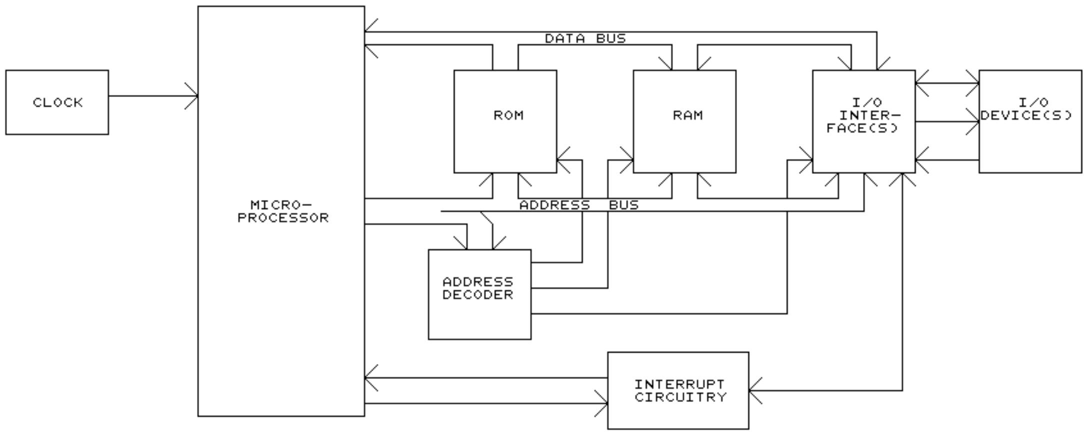


Fig. 1-2. Typical computer block diagram.

computer can read and use these programs, but cannot erase or change them.

The RAM

The SK68K computer's RAM consists of two parts - static RAM, and dynamic RAM.

Most computers would generally have either static RAM (abbreviated SRAM) or dynamic RAM (DRAM), but not both. We use both because they each have their advantages and disadvantages. For large memories, dynamic RAM is cheaper and smaller - without it, it would not be practical to provide 1 megabyte of static RAM at a reasonable cost. On the other hand, for small memories static RAM is the right choice because it is much simpler - and easier to debug in case of problems! Hence the SK68K computer will first be built with a small amount of static RAM using just two integrated circuits. Since the static RAM circuitry is so simple, it will probably work immediately without any problems, giving us the ability to run Basic and HUMBUG. Once the static RAM is all working, then we can add the dynamic RAM, consisting of thirty-two 256K dynamic RAM ICs plus a batch of support ICs. If there is a problem, we can use HUMBUG to debug the dynamic RAM. This kind of bootstrapping makes the building of a large system like the SK68K from scratch a lot more practical.

There is actually an ulterior motive to providing static RAM - to provide a clock/calendar we need only unplug one of the RAM ICs and substitute a clock/calendar IC which is totally pin compatible. We will use the MK48T02 which provides not only a clock and calendar, but also some static RAM of its own - and a built-in battery to keep the clock and RAM going while the computer is turned off.

I/O Interfaces

Although Fig. 1-2. shows just a single box labelled I/O Interfaces, the SK68K computer's I/O is actually quite complex. It consists of two MC68681 DUARTs to provide four serial interfaces, one 68230 parallel interface/timer, a 1772 floppy disk controller, keyboard interface, speaker interface, a number of extra support ICs, all of the circuitry needed to interface to the six PC-compatible interface connectors, plus the interrupt circuitry shown at the bottom, which allows I/O devices to interrupt the 68000 when they need it; the latter is absolutely essential for using a clone keyboard.

Some microcomputers often also provide DMA or Direct Memory Access circuits. This is a feature which is often used when the CPU has difficulty keeping up with fast I/O devices such as disk drives. Since the 68000 does not have any problems keeping up (and DMA really complicates the computer), we chose not to use it in the SK68K computer.

The Data Bus

As Fig. 1-2 shows, the two main sets of connections between the microprocessor and the ROM, RAM, and I/O interfaces, are the data bus and the address bus. The term bus is used to signify that a number of parallel wires are used to carry data simultaneously.

The data bus is used to move numbers (which could be numeric data, instructions, or text) between the microprocessor, memory, and I/O devices. If you look at the arrowheads at the ends of the data bus in Fig. 1-2, you will see why the data bus is said to be bidirectional. Data may go in either direction - left or right.

In our case, the data bus consists of 16 wires, each of which carries one bit or binary digit. (See Appendix B if you are not that familiar with binary numbers and bits.) Thus the 68000 can transfer a 16-bit number to or from the microprocessor all at once. As we will see, however, the 68000 handles numbers in chunks of 8 bits (called a byte), 16 bits (two bytes, also called a word), or 32 bits (four bytes, also called a long word.) When transferring a byte, the 68000 uses only half of the data bus; when transferring a long word, it uses the data bus twice, transferring 16 bits at a time.

The number of bits on a data bus - also called the width of the bus - obviously has a bearing on the speed - the wider the bus, the more bits can be moved at a time, so the faster the computer runs. But there is more to the story - the size of numbers that can be handled internally in the microprocessor is also important.

The very first general-purpose microprocessors - the 8080, 6800, 6502, and Z-80 - had an 8-bit data bus and also handled 8-bit numbers internally. For this reason these were called 8-bit microprocessors.

The next generation of chips, such as the 6809 and 8088, still had 8-bit data buses, but could now handle 16-bits internally. This gave them extra power, but they were still bogged down by the slow speed at which they could transfer data to and from memory and I/O devices.

The next step up included the 8086, 80186, and 80286, processors which could handle 16-bit numbers both internally and externally, and which are called 16-bit processors.

The 68000 is one step higher yet - it still only has a 16-bit data bus, but can handle 32-bit numbers internally.

Finally, at the top of the current pyramid are the 80386 and 68020, both of which handle 32-bit numbers both internally and on the data bus. These are true 32-bit processors.

But even this is not the entire story - there are still other factors which affect computer speed. Though there are processors which have a wider data bus than the 68000, a bus that's twice as wide doesn't necessarily mean a computer that's twice as fast unless you consistently run programs that make full use of that width. What does make the 68020 and 80386 faster than the 68000 or 8086 is their more extensive use of a cache. This is an area of memory within the processor that holds instructions or data that are read out of memory before they are needed. Whereas older processors would only read data out of memory at the instant it is needed - and then have to wait for it - newer processors may spend their spare time pre-reading a few bytes ahead of themselves, and store the bytes read just in case they should be needed in the next few instructions. Alternatively, they may store instructions that have been recently used in case they are needed again soon. In this way they avoid the need to wait for data or instructions to come in from memory. The 68000 does have a small cache, but it is too small to provide a significant saving.

The Address Bus

The other major bus, the address bus, carries addresses. That is, in order to save data into memory, or read data from memory, the processor must specify exactly where in memory that data is located. This is done with a numeric address, sent out on the address bus. As you can see from the arrowheads in Fig. 1-2, the address bus carries data from left to right; that is, it is unidirectional. (There is an important exception - when a computer uses DMA, then addresses may come out of an I/O interface and travel to the left.)

The width of the address bus determines exactly how much memory a computer can have. If the bus had only three lines, for example, then each address would consist of just three bits. Since each bit can only be either a 0 or 1, there would be just eight possible addresses - 000, 001, 010, 011, 100, 101, 110, or 111 - since there is no other three-bit number that can be made out of ones and zeroes. Hence the maximum number of addresses - or put another way, the maximum possible number of locations in the memory of this computer - would be eight, which also happens to be equal to two to the third power. That is, $2^3 = 8$.

In general, the maximum number of addresses is 2 to the same power as the number of address lines. For example, most 8-bit computers have 16 address lines in their address bus, so they have a maximum of $2^{16} = 65536$ addresses.

Since a K in computer terms is 1024 (not 1000 as in ordinary electronics), 65536 works out to be exactly 64K locations.

Newer microprocessors have more address lines than their predecessors:

microprocessor	address bus width (bits)	maximum memory size
8080, 6800, etc.	16	64K
8088, 68008, etc.	20	1 megabyte
68000	24	16 megabytes
68020, 80386	32	4 billion bytes

As you might expect, there is more to the story than just the width of the address bus. Consider the 20-bit bus of the 8088 and 68008, for example. Both of these processors can address up to a megabyte of memory, but the 68008 (the smaller cousin of the 68000) can do so in one continuous piece, whereas the 8088 must split that memory into 64K segments. Handling the segmenting greatly complicates a program - that's why many programs written for the 8088 (such as Microsoft's BASIC or BASICA) can only use 64K of memory at a time, whereas a Basic on the 68008 has no such limitation.

Thus the 68000 can easily handle programs and data that use up the entire 16 megabytes of memory ... almost. There is a difference between the way that Intel and Motorola processors handle I/O. In a computer using a Motorola processor like the 6800 or 68000, the I/O interface connects to the processor in exactly the same way as the memory, with the result that memory and I/O use the same addressing scheme. Thus if the 68000 were to use 1 megabyte to address I/O, then there would only be 15 megabytes left for memory. Intel processors do not have that limitation - they use the entire normal address range for memory, and have a separate set of addresses (usually much smaller) just for I/O. Although some people point this out as a weakness in the Motorola approach, in practice it makes very little difference since I/O seldom requires more than just a few dozen (or perhaps a few hundred) addresses. There are still plenty of addresses left for memory. In most cases, a 68000 or 68020 has so many possible addresses that we can afford to waste thousands - maybe even millions - of addresses on I/O without feeling the pinch.

A list of addresses in a computer and what they are used for is called a memory map. Table 1-1 shows a simplified memory map of the SK68K computer. 1 As you can see, there is still plenty of memory left for expansion, probably a lot more than most of us would care to pay for.

Table 1-1. Simplified SK68K Computer memory Map

Memory Range (hex)	Description
000000 - 0FFFFF	Dynamic RAM (1 megabyte)
100000 - BFFFFF	Empty - for expansion (11 megabytes)
C00000 - DFFFFF	Addresses for PC expansion slots (2 megabytes)
E00000 - F7FFFF	Unused (1.5 megabytes)
F80000 - F9FFFF	ROM (128K)
FA0000 - FBFFFF	Addresses for PC expansion slots (128K)
FC0000 - FDFFFF	Unused (128K)
FE0000 - FE3FFF	I/O Interfaces (16K)
FE4000 - FEFFFF	Unused (48K)
FF0000 - FF7FFF	Static RAM (32K)
FF8000 - FFFFFF	Unused (32K)

The Address Decoder

As Fig. 1-2 shows, the address bus coming out of the microprocessor is split into two parts - part goes into the address decoder, while part goes to the ROM, RAM, and I/O interfaces.

The job of the address decoder is to look at the address on the bus and decide whom it's intended for. For example, as Table 1-1 shows, the dy-

1 Table 1-1 shows memory assignments, but some of these may not be used. For example, 32K is assigned to static RAM, but only 4K is actually installed.

namic RAM occupies addresses 000000 through 0FFFFF. Whenever the address decoder sees any address beginning with the hexadecimal digit 0, it recognizes it as a RAM address, and sends a signal to the RAM that effectively says “Hey, you! This address is meant for you ... go to work." This signal is called an enable or select signal. If it goes directly to an IC, then it is called a chip enable or chip select, often abbreviated CE or CS.

Fig. 1-2 shows just one address decoder, connected to the ROM, RAM, and I/O interfaces. In practice, though, most computers split that address decoder into two or more smaller decoders, each of which services just one part of the computer. Part of the reason is that it is easier to build that way, but there is a second reason as well - not all the decoders look at the same part of the address bus.

In the case of dynamic RAM, the address decoder need only look at the leftmost hex digit of the address; that is, it looks at the four leftmost bits, which must equal 0000 (a hex 0) for the RAM to go to work (see Appendix B for a discussion of binary and hexadecimal digits if you need to brush up.)

The decoder for the ROM, on the other hand, must look at seven bits. As Table 1-1 shows, the ROM occupies addresses F80000 through F9FFFF. Since there are other parts of the computer whose addresses also begin with the hexadecimal digit F, the ROM’s address decoder must look at more than just the first digit F - it must also check that the second digit is either an 8 or a 9. This is done by looking at individual bits of the address.

When written in binary, the lowest ROM address - F80000 - begins with the bits 1111100 and then continues with 17 zeroes, like this:

F80000 = 1111 1000 0000 0000 0000 0000.

The address F9FFFF also begins with 1111100 but then continues with 17 ones, like this:

F9FFFF = 1111 1001 1111 1111 1111 1111.

All other ROM addresses also begin with the bits 1111100, but have different combinations of 17 zeroes and ones at the end. Thus any address which starts with the bits 1111100 applies to the ROM; the ROM’s address decoder therefore looks for a 1111100 bit pattern in the first seven bits of the address, and sends an enable signal to the ROM as soon as it sees it.

Hence different parts of the address decoder look at different bits of the address bus. Some parts may only look at one or two bits, other parts may look at four or six, and some parts of a typical computer’s address decoder may look at 16, 32, or even more bits in some computers.

Conclusion

We conclude this chapter by just reviewing that the typical computer consists of a CPU (called a microprocessor in a micro computer), some memory (both ROM and RAM), I/O devices and their I/O interfaces, and various other circuitry made up of glue chips. And let us not forget the subject of the next chapter - the power supply.

Chapter 2

The Power Supply

An unreliable power supply can play havoc with a computer. Yet the power supply is so common - and inconspicuous - that it is taken for granted and seldom suspected in case of a problem. Since nothing else can work without the power supply, let us start with it.

2-1. Discussion

The SK68K requires three power supply voltages. But it is difficult to say exactly how much current it needs at any specific time, since this depends on how far along you are in constructing it, and also on how many plug-in expansion boards are installed in the six XT-compatible connectors.

A fully configured SK68K, with a full 1 megabyte of memory and all on-board options, requires the following:

+5 volts for the main TTL and MOS logic.

+12 volts for the RS-232C serial port, and for some of the I/O circuits.

-12 volts for the RS-232C serial port, and for some of the I/O circuits.

These voltages can be supplied from one or more regular power supplies, but the simplest and cheapest is a 135-watt or 150-watt switching power supply of the type designed for XT clones.

Types of Power Supplies

Switching power supplies are substantially smaller and more efficient than the old-fashioned “linear" power supplies. The block diagram of Fig. 2-1 shows the difference between the conventional linear supply and a switching supply.

The conventional linear supply begins with a step-down transformer, which steps the 115 volts AC from the power line to a more manageable voltage, slightly above the desired DC output voltage. The resulting AC is

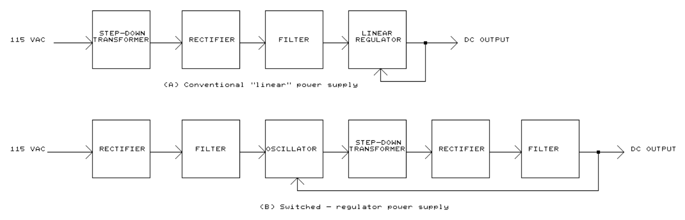


Fig. 2-1. Types of power supplies.

then full-wave rectified, filtered, and then fed to a linear regulator. The most common regulator contains a pass transistor in series with the output, controlled by a voltage comparator which compares the DC output voltage against a reference voltage. The comparator then biases the pass transistor to change its resistance; it thus uses the voltage drop across the transistor to keep the output DC voltage constant. If the output voltage is too low, it makes the pass transistor conduct more; if the output voltage is too high, it biases the pass transistor so it conducts less.

There are three main problems with such a linear circuit:

(1) Since the AC line operates at 60 Hz, the step-down transformer must be fairly large and heavy. It must have enough iron in the core so as not to saturate at the low frequency.
(2) The filter capacitor must also be fairly large so as to remove the ripple, which occurs at twice the line frequency. Although the capacitor need not remove all the ripple (since the regulator can remove the rest), it must remove enough ripple to make sure the voltage fed to the regulator does not drop too far.
(3) The regulator's pass transistor conducts current at all times, and therefore dissipates power. To safeguard against the voltage dropping too far, the voltage level into the regulator must be at least several volts more than the desired output, so the power dissipated by the pass transistor may be considerable.

None of these problems is major, but it does mean that a linear power supply to provide a substantial amount of power must be fairly large and heavy, and must dissipate a substantial amount of power.

The switched-regulator supply of Fig. 2-1 (B) is substantially more complex. It starts with a rectifier and filter, which directly change the incoming 115 volts AC into DC at between 110 and 150 volts. This DC then powers an oscillator, which generates a high voltage AC at a frequency of several kHz. This signal is then stepped down through the transformer, rectified, and then filtered.

Regulation is achieved by again comparing the DC output of the supply against a reference voltage in a voltage comparator, and using the resulting output to control the pulse-width of the oscillator.

This circuit has several advantages over the linear supply:

(1) Since the transformer works at several kHz, rather than at 60 Hz, it requires a smaller iron core. A small toroid can be used with less loss and less cost.
(2) Likewise, the final filter capacitor can be small since the ripple frequency is much higher than in the linear supply. (Although a second filter is

needed in the input circuit, the oscillator will tolerate a large amount of ripple and thus a small capacitor can be used here as well.)

(3) Although the oscillator transistors indirectly provide regulation of the output, they do not operate in the linear region as the pass transistor would in a linear power supply. They are always either cut off or saturated, and so their power dissipation is much lower than in the linear region.

The result is that a switching power supply, although much more complex than a linear supply of the same output power rating, is generally much smaller and lighter, and runs much cooler. For example, a 135-watt or 150-watt power supply of the type often used in XT clones weighs only a fraction as much as an equivalent supply used to weight before switching power supplies became popular.

Needless to say, switching supplies do have some disadvantages. The major ones are this:

(1) Because the oscillator operates at fairly high powers and high frequencies, it can produce interference with nearby radio or television receivers. The 115-volt input to the power supply must therefore be well filtered to prevent high frequency signals from being transmitted back into the power supply; additional filtering is also needed on the dc outputs as well, and the supply must be well shielded.

(2) Since the supply does not have a large output filter capacitor, it does not do well when the output current suddenly changes. In a conventional power supply, the output capacitor easily handles sudden surges in power; transient response of switching supplies tends to be slower. The lack of a large filter capacitor also means that switching power supplies are also much more susceptible to short power outages. For example, some users may use a UPS (Uninterruptible Power Supply) or standby power supply in case of main AC power failure. Such power supplies often delay a anywhere from 15 milliseconds to as much as 1/2 second after a main power failure before they switch in to provide power. The large filter capacitor in a linear power supply may be able to provide power to tide the computer over during this interval; in a switching power supply, there is no such storage capacitor and so the dc power may just totally disappear for this fraction of a second.

(3) Multi-output power supplies (such as an XT supply which provides four different output voltages) often derive all their output voltages from the same regulator and transformer. In other words, the switching regulator in an XT power supply controls all four outputs at the same time. Typically, it regulates only the +5-volt output, while the other three outputs are allowed to vary somewhat. For example, if the load on the +5-volt output goes up, the switching regulator increases the output of all four supplies at the same time. The +5-volt output may thus stay constant, but the other three outputs rise. Some power supplies try to compensate by providing low-power linear regulators on the other outputs, but this obviously limits their output capacity. In most XT-type power supplies, the fan is powered from the +12-volt supply; you can

often hear it speed up or slow down slightly as the switching regulator changes its pulse width to keep the +5-volt output constant.

(4) Switching regulators do not work very well if there is no load. Most XT-type power supplies therefore have a protection circuit which turns off the entire power supply when the load is removed; the same circuit also turns off the supply if the dc output is shorted.

SK68K Power Supply Connector Pinouts

The XT-type power supply has six dc output connectors, two for the electronics and four for disk drives.

Two six-pin connectors plug into the main computer board (J10A and J10B in the right rear corner of the board). The only reliable way to differentiate between them is by the colors of the wires. In order, their 12 connections are as follows:

Pin number	Color	Function
		J10B
1	Orange	Power Good (not used - see text)
2	-	No Connection
3	Yellow	+12 volts DC
4	Blue	-12 volts DC
5	Black	Ground
6	Black	Ground
		J10A
1	Black	Ground
2	Black	Ground
3	White	-5 volts DC (not used)
4	Red	+5 volts DC
5	Red	+5 volts DC
6	Red	+5 volts DC

These connectors provide two outputs which will not be used by our computer - a “power good" output which could have been used to detect that primary (115-volt) power has just disappeared and the dc outputs are also about to disappear, and a -5-volt output which is simply not needed.

The remaining four power supply connectors are all wired identically as follows:

Pin number	Color	Function
1	Yellow	+12 volts DC
2	Black	Ground
3	Black	Ground

Pin number	Color	Function
4	Red	+5 volts DC

5-1/4" floppy and hard disk drives all use the same wiring, so such a power supply can directly power as many as four such drives (assuming that it has enough power handling capacity.) 3-1/2" drives use a different connector and pinout, and an adapter cable has to be used.

2-2. Construction

Before proceeding with actual work, it's important to set things up so the SK68K printed circuit board can be worked on easily, yet is protected from accidental short circuits and other possible damage.

The best way to do so is to is to mount the board and its power supply on a wooden board about 12" by 24", as shown in Fig. 2-2. Hammer two brads into the board as shown to hold the printed circuit board in place. Be sure to use the correct two holes to avoid a possible short circuit. (The location of the two brads in Fig. 2-2 is shown with the small triangular flags on the brads.)

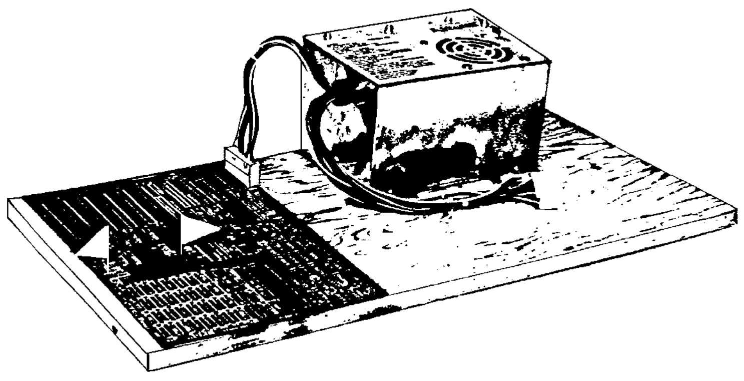


Fig. 2-2. One way to mount the board and supply.

Note how the board is oriented - power connector J10 is right next to the power supply, and the six expansion connectors are in the left rear corner. We will use the words left, right, front, and back to describe the board when it is positioned like this (it will fit into a PC clone cabinet the same way). For example, U92 is in the back, while C47 is in the front left corner. (Look at the silk-screen printing on the board to see how the components are positioned on the board.)

Note also that the side with all of the white lettering - this is called the silk-screen layer - is on top, whereas the other side of the board will be called the bottom. All of our soldering will be on the bottom side -there are no solder joints whatever on the top or silk-screen side of the printed circuit board.

Soldering itself is more of an art than a science. Even if you consider yourself to be an expert, read Appendix C for our hints on how to keep bad soldering from ruining your SK68K project.

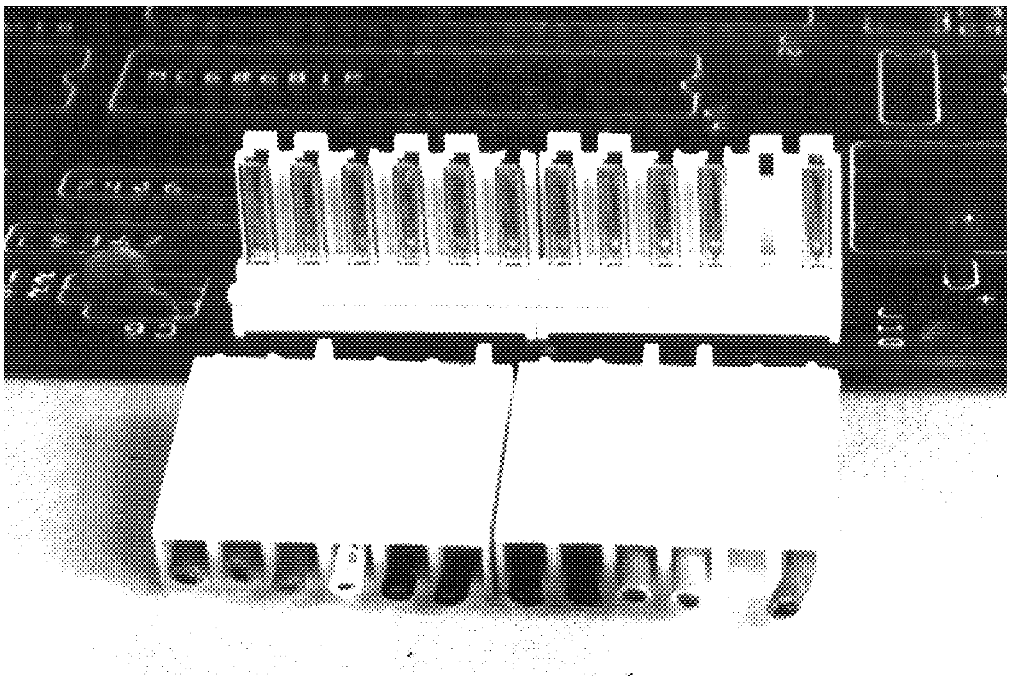


Fig 2-3. Power connectors and plugs. Note how the tabs match

The Power Connector

The power connector, J10, actually consists of two six-pin connectors, J10A and J10B, in the right rear corner of the board. They are shown in Fig. 2-3, J10A on the left and J10B on the right. Read the following paragraphs before you do anything.

The power connectors are a potential source of big problems. If you look at the connectors you have, you will note that the two board-mounted connectors are identical, and the two power supply plugs are probably identical as well. In other words, it is extremely easy to make a mistake and plug the wrong power supply plug into the wrong connector on the board and burn up the board. We have to make sure that never happens.

First, look at the two power supply plugs. You will see that one of them has six wires, while the other has only five - the next-to-the-last wire is missing. Note also that we have cut off the next-to-the-last pin on J10B in Fig. 2-3 (compare it with your connectors). Though this doesn't really prevent a mixup, it does serve to remind us what goes where.

Next, compare the tops of J10A and J10B in Fig. 2-3 with the actual power connectors in your parts kit. In the plastic, behind each of the metal pins, is a small rectangular opening with a tiny plastic 'bridge' above it. Your connectors will still have all of these bridges, while some of the bridges are shown cut off in Fig. 2-3. Now look at the two matching power supply plugs, which will have six small plastic tabs sticking out the long side. These tabs may all still be there, or some of them may already be cut off. When all the plugs and connectors are brand new, the tabs on the plugs prevent them from being inserted into the pc-mounted connectors because the long tabs hit the bridges. The object is to cut just the right combination of tabs and bridges so the six-wire plug only fits J10A, and the five-wire plug only fits J10B. If you look closely at Fig. 2-3, you will see that we have done exactly that. (The whole thing is complicated by the fact that the power supply plugs may already have some tabs cut off, so you may have to take that into account.)

One useful piece of information: when properly installed, the black wires of the two connectors are adjacent to each other.

Now that you know what has to be done, install the following components:

J10A and J10B	Solder the two connectors to the board as in Fig. 2-3, and then match up the bridges and tabs so the power supply plugs in only one way. Make sure that the connectors are oriented the correct way, so that the metal pins are visible from the edge of the board, as in Fig. 2-3, and that the 5-wire plug only fits J10B.
C65	10 µF tantalum capacitor; make sure that its positive lead (marked by a + sign) is closer to J10, as tantalum capacitors have a nasty habit of exploding if connected backward!
C6	0.1 µF disk capacitor near J10
C3, C4, and C5	47 pF disc ceramic capacitors
C68	33 pF disc ceramic capacitor

Although not part of the power supply, the 47 pF and 33 pF capacitors are very similar to the many 0.1 µF capacitors, and this gets them out of the way so you will not confuse them later.

Aside from one 1 µF tantalum capacitor, all the remaining capacitors are 0.1 µF disc ceramics. Digital circuits are notoriously ’noisy’, and computer designers have learned the hard way that it is necessary to install small bypass capacitors between the +5-volt line and ground all over a board to keep that noise off the power lines. A general rule of thumb is that one such capacitor should be installed for every two or three digital ICs. We will instruct you when to install these additional capacitors.

2-3. Testing

We will do no testing at this stage; the power supply will be tested in the next Chapter.

Chapter 3

LED Indicators

First, a bit of theory and some terms we are going to need later. Even if you know all about logic circuits, read on - some of these concepts are a bit different in real life from the way they are sometimes written up in simple books and magazine articles.

3-1. Discussion

Digital circuits represent the binary digits 0 and 1 by means of voltages; in most microcomputers, the two voltages are often called low (which is a voltage between 0 volts and roughly 0.8 volt) and high (which is a voltage between about 2 volts and 5 volts). There are a few exceptions, of course - such as in an RS-232 circuit between a computer and terminal where larger positive and negative voltages may appear - but lows near 0 volts and highs near 3 to 5 volts are the most common. In any case, the range between 0.8 volts and 2 volts is a no man's land; if a digital signal is in that range it usually indicates a problem somewhere.

Many people think that a low voltage is a 0, while a high voltage is a 1, but this is not always true - it could be the other way around. So talking about ones and zeroes can be ambiguous, while talking about lows and highs is always quite specific. Note that we don't really care about the exact value of a signal's voltage, so long as it falls into one of these two ranges.

But we can talk about digital signals in a different way as well - we can say that a particular signal is on, or off. Computer people, however, like somewhat longer words - they say that a signal is asserted when they really mean it is on, and they may say that it is negated when it is off.

Now comes the problem - some circuits use a high to mark a signal as on (asserted), while other circuits may use a low to turn on (assert) a signal. So we run into two types of circuits:

A so-called active high circuit is one which is high when asserted (on), and low when negated (off); some books call this positive logic. A so-called active low circuit is one which is low when asserted (on), and high when negated (off); some books call this negative logic. In a typical computer, both kinds of circuits may be used, and often an active high circuit may be just a tenth of an inch from an active low circuit.

Many of the signals in this text and diagrams are assigned meaningful names. Whenever you see a name which has a “not bar" either above or below its name, such as $\overline{\text{HALT}}$ you will know that this signal is active low. On the other hand, a signal without the not bar, such as FC0 or A16, is active high. Because it is difficult to place not bars over signal names in text, many people use alternative ways of marking active low signals; some common ways are with an asterisk, as in HALT*, a minus sign, as in -HALT or HALT-, or with a lower case letter n, as in nHALT. We will use the not bar above the name in this book.

LED Indicators

If, instead of using an XT-clone cabinet, you use what is commonly called a “mini-AT" cabinet (because it is the size of an XT cabinet, but is built in the style of an AT cabinet), you will note that such cabinets have two or three status indicator LEDs on the front panel. These LEDs can eventually connect to J15, J16, and J17 on the SK68K board as shown in Fig. 3-1 (which also shows the speaker wiring.) In each case, a resistor in series with the LED (or speaker) limits the current through it, while the LED (or speaker) is controlled by a section of U32, a 7406 open collector gate. (Note that U32c, U32f, U32d, and U32b are all part of the same U32 IC; U32 has six such inverters, and the other two are used elsewhere. The numbers on the

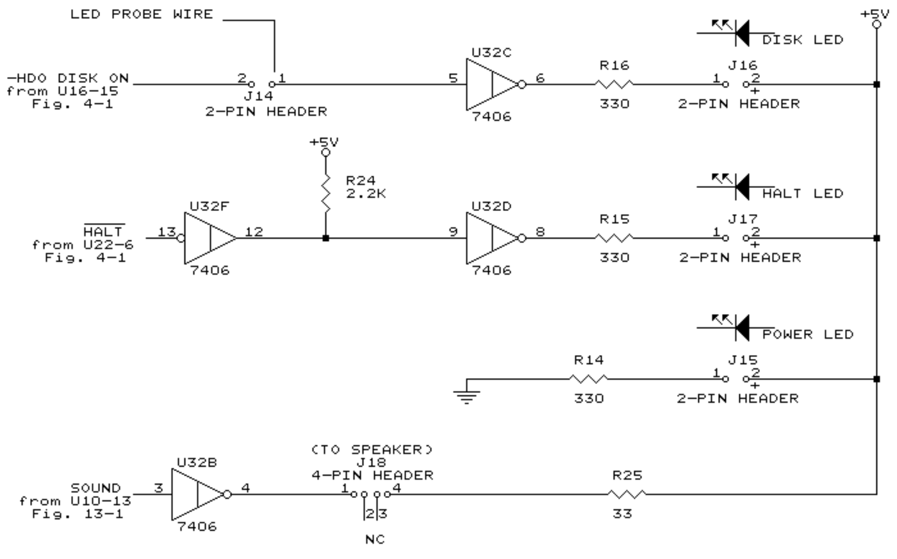


Fig. 3-1. LED and speaker circuit.

outside of the triangle, next to the leads, are the pin numbers. The vertical line inside the U32 symbol marks this as an open collector device, explained shortly. Ignore the fact that U32f has a small circle, called a bubble, on its input side instead of the output; this is just a different notation and will be explained next time shortly.

The LED at J15 lights whenever there is +5-volt power, while the LED at J17 lights when the 68000 is halted, and the LED at J16 lights to indicate hard disk use.

3-2. Construction

Since many users of the SK68K do not have access to an oscilloscope or even a logic probe, the wiring for the LED at J16 has been set up to allow its use as a simple logic probe. Furthermore, during construction you will not have the printed circuit board mounted in the cabinet, and so we want to connect LEDs directly to the board for immediate use. (This is especially important for the HALT LED, which will be very useful during checkout of the board.)

We therefore wire the circuit a bit differently. Refer to Fig. 3-2, the parts layout, and install the following parts:

R14, R15, and R16	330 ohm 1/4-watt resistors
R24	2200 ohm 1/4-watt resistor
C11	0.1 µF disk ceramic capacitor
	14-pin socket for U32
R25	33 ohm 1/4-watt resistor
J18	4-pin header strip

Do not install U32 in its socket yet. (R25 and J18 are not needed yet, but this is a convenient time to install them as the speaker wiring is so similar to the LED wiring. But do not connect the speaker yet.)

Then install the three LEDs at J15, J16, and J17. The negative lead of each LED, usually marked by a small flat on the side, should go toward the resistors. If at all possible, check each LED first, since many times LEDs available on the open market are wired opposite to this convention.

Install each LED so it stands up straight, but the bottom of the LED is about 1/2" above the board. (The reason: When we're ready to mount the board in the cabinet, we will cut off each LED lead just below the LED itself, and use the stubs of the LED leads as connectors for the panel-mounted LEDs.)

3-3. Testing

Now connect the power supply to J10 and power up the board. The POWER LED should light, though it may immediately go off again. If so, don't be alarmed - most PC-type power supplies shut themselves off if there is not enough of a load on them, and a single LED is a very small load indeed. Simply turn off the supply, temporarily connect the 150- or 330-ohm

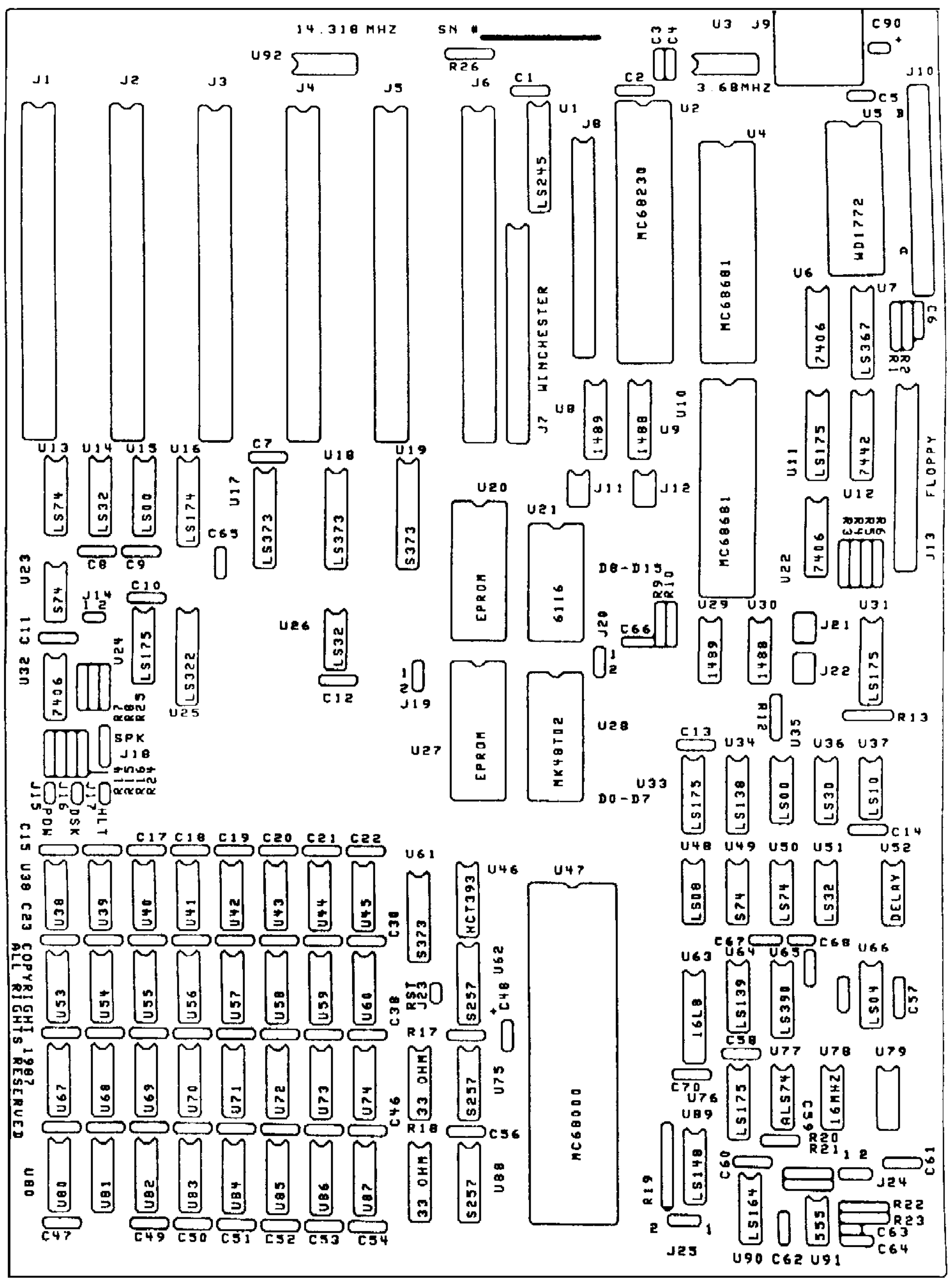


Fig 3-2. Printed circuit board layout.

resistor between pins 7 and 14 of the U32 socket (don't force the leads into the socket) and try again. This should add just enough of a load to allow the supply to turn on.

If the LED does not light at all, even for an instant, then most likely either the LED is in backward, R14 is the wrong value, or the power supply is defective or not properly connected to J10A and J10B. Correct the problem before continuing.

NOTE:

> During construction, we will often wire something, turn on the power and try it out, turn off the power, wire some more, and so on. It is absolutely essential that you turn off the power before doing any wiring, soldering, or inserting ICs into sockets. Better yet, turn off the supply and also unplug it. If you slip and forget to turn off the power, you may well burn out part or all of the components on the board, and perhaps even burn out a few of the traces as well.

So now turn off the power, connect a thin wire about 12-15" long to terminal 1 of J14. Try to use a thin solid wire, about 30 gauge. If you use a stranded wire, then twist the strands of the loose end and cover them with a bit of solder so they stick together. Next, insert a 7406 IC into U32 (remove the 150- or 330-ohm temporary resistor). Note that all ICs on the entire board are oriented the same way - pin 1 (marked by a dimple or notch, both on the IC and also on the silk screen layer on the board) goes toward the back of the board. Then turn the power back on.

The wire connected to J14-1 (which is shorthand for terminal 1 of J14) is now a test probe, which we will call the LED probe. If you ground its loose end (to pin 7 of IC32, for example) then the LED at J16 should go off; if you connect it to a high voltage (pin 14 of IC32, for instance) then the LED should go on. The J16 LED now makes a simple logic probe which can be used to check out other parts of the computer. (If you have a meter, oscilloscope, or real logic probe, then feel free to use it instead, but you may still occasionally want to use this built-in probe instead.)

When the LED probe wire is connected to a low or ground, the LED will be dark; when connected to a high or +5 volts, it will be brightly lit. When it is not connected to anything at all, then the LED will be on, for the simple reason that TTL ICs see a disconnected input as if it were high. When connected to a source of pulses, the LED will light, but its brightness will depend on the type of pulses - a pulse signal which is high most of the time will be brighter than one which is mostly low. If you connect the LED probe to a pulse signal, the LED will usually dim slightly from its normal bright light (because of the open circuit); this is an easy way to recognize a pulse signal.

Chapter 4

The Reset Circuit

The 68000 microprocessor must be initialized when the system is first turned on, or whenever it must be restarted from a major error. This process is called *resetting.*

4-1. Discussion

Resetting is done by temporarily grounding two 68000 pins - the $\overline{\text{RESET}}$ line and the HALT line. Remember - the “not bar" denotes that these signals are active low. Hence grounding them, which forces them to a low, asserts these two lines or turns them on. Asserting $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ together for a minimum of 100 milliseconds resets the 68000 and gets it ready to run a program.

The 68000 should be automatically reset every time the power is turned on, but it is also useful to have a button which can be pushed to force a reset if the computer does something it is not supposed to do. Both of these functions are done with the circuit of Fig. 4-1.

The main IC in the circuit is U91, a 555 timer which is connected to a timing circuit consisting of R23 and C63. When the computer is running, C63 is charged through R23 to about +5 volts, and then the output on pin 3 of the timer is a low; this is inverted by the two U22 inverters to a high. Actually, this description is not entirely correct. U22 is a 7406, which is an open collector (or o.c.) hex inverting buffer. Open collector devices (marked on diagrams by a vertical line inside the logic symbol) are missing the part of the output circuit which can output a high; hence they can only output a low or nothing. In this case, they output nothing - an open circuit. But because of R20 and R21, two 2200-ohm resistors connected to +5 volts, the RESET and $\overline{\text{HALT}}$ lines are pulled high by the resistors instead; that’s why these resistors are called pullups. In general, if you ever see an open-collector device which does not have some sort of a pullup resistor connected to

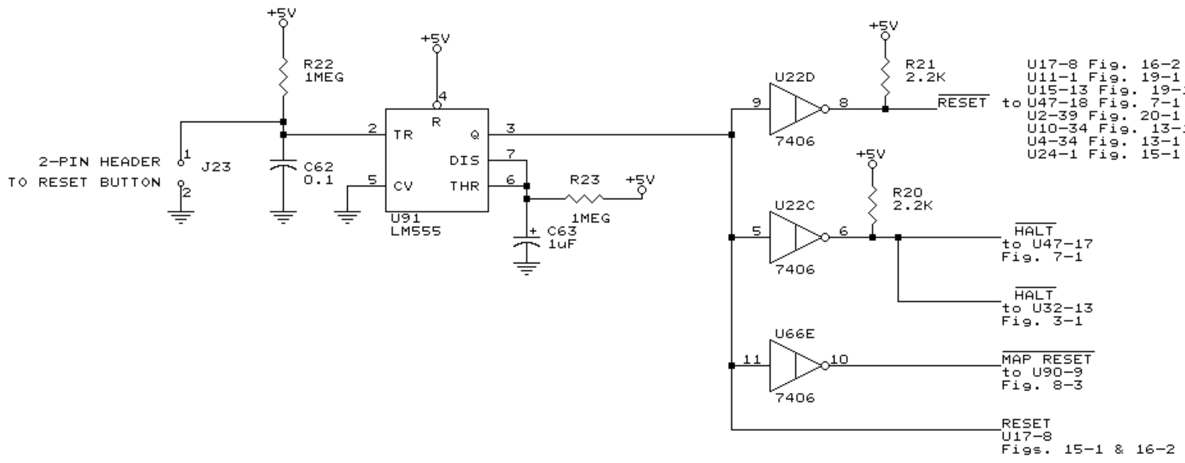


Fig. 4-1. The RESET circuit.

its output (the LED circuit in Fig. 3-1 was a pullup in a way) it usually means somebody made a design error.

Whenever a pushbutton connected to J23 is pressed, this applies a low to pin 2, the trigger input of the timer, which causes the timer to ground pin 7, which discharges C63. (This also happens when power is first applied, since C63 would normally start off discharged). The 555 timer sees this low voltage on its pin 6, and therefore outputs a high on pin 3. This is inverted to a low by U22, and asserts a low on RESET and HALT of the 68000, resetting it. (The reset signal also goes elsewhere through U66e, but more on that later.)

As soon as the pushbutton is released (or the power supply voltage has risen), C63 starts to charge through R23. When it reaches about 3.3 volts, the 555 timer senses this rise and shuts off its output on pin 3; this removes the low from RESET and HALT, and lets the 68000 begin operating.

How long does it take for the voltage on C63 to reach 3.3 volts? About one time constant, which is defined as the product of R23 and C63. Since R23 is 1 megohm (1,000,000 ohms) and C63 is 1 µF (0.000001 farads), the product is 1,000,000 x 0.000001 = 1 second. Thus the RESET and HALT signals will go low for about 1 second at startup or whenever we push the reset pushbutton.

4-2. Construction

Install the parts listed below, but note that tantalum capacitor C63 is a polarized capacitor; its positive terminal must go toward pin 6 of U91. Also, the two-pin header strip for J23 has a short end and a long end; the short end goes through the board and is soldered on the bottom, while the long end sticks up.

R22 and R23	1 megohm 1/4-watt resistors

R20 and R21	2200 ohm 1/4-watt resistors
C57, C61, C62 and C64	0.1 µF disc capacitors
C63	1 µF 16-volt tantalum capacitor
J23	a two-pin single header strip
U91	555 timer and its socket
U22	7406 open-collector buffer and its socket
U66	74LS04 hex inverter and its socket
	Two unmarked 0.1 µF capacitors to the left of U66.

4-3. Testing

Turn on the power. The HALT LED should go on for about a second, and then suddenly switch off.

Now use the LED probe wire connected to J14 to check the signals at the outputs of U22d, U22c, and U66e. Connect the probe to one of these, and use a screwdriver or wire to short the two pins of J23; the test LED should go off and then, a second or so later, back on, indicating that the signal went low and then back high.

Chapter 5

The Master Clock Circuit

The “clock" circuit of a computer is actually not a clock in the traditional sense (since there is a separate section called a “clock / calendar". Rather, it is an oscillator which is more like a metronome or drill sergeant. It supplies pulses which keep all the parts of the computer marching in step.

5-1. Discussion

Fig. 5-1 shows the diagram of the master clock for the entire computer. U78 is a 16 MHz oscillator module containing a crystal oscillator and all the logic circuitry to provide a square wave output at the right levels for TTL logic circuitry. Its output goes to U77a, a 74ALS74 type-D flip-flop wired as a divide by 2. Each time the CK (clock) input goes from a low to a high, the flip-flop flips from one state to the other. Its output therefore goes through a complete cycle once for every two input cycles, so its output is at 8 MHz, exactly half of the 16 MHz input. This signal, called CLK8, is used in a number of places throughout the computer.

In addition, if J24 has a jumper from the center terminal to terminal 1 (which would be the normal situation), U77b also divides the 16 MHz by two and provides an 8 MHz clock signal, called MPUCLK, to the 68000 and elsewhere.

To run the computer at 10 MHz, you would install another oscillator module, running at 20 MHz, at U79 and place the J24 jumper in position 2. CLK8 would still be at 8 MHz, but MPUCLK would now run at 10 MHz. Two modules are necessary because CLK8 is used elsewhere in the computer and must stay at 8 MHz even if the 68000 itself runs faster.

Incidentally, the small triangle inside the clock inputs on U77 indicates that these inputs respond to a change of voltage, also called an edge. Since there is no bubble on the outside of this pin, the clock input responds when the input goes high (i.e., a positive edge.)

5-2. Construction

Now mount the following components:

U78	16 MHz oscillator and its special socket. Note that three corners are rounded; the pointed corner identifies pin 1
U77	74ALS74 (ALS, not LS) and its socket
J24	3-pin header
C58, C59, C60	0.1 µF disc capacitors
	a shorting jumper from the center pin to pin 1 of J24

Although U78 is installed in a socket, in most applications it would be soldered directly to the board. Note that a special socket is needed since its pins are round, whereas most IC pins are rectangular.

5-3. Testing

Next, power up the computer. If you have an oscilloscope or a logic probe which can detect pulses, test the CLK8 and MPUCLK lines for the required pulses (inexpensive oscilloscopes may have trouble displaying the clock pulses, or may show them as a very distorted sine wave.)

Testing is a bit tougher if you only have the LED probe connected to J14-1; still, it can be done. First, note how bright the LED is when the probe wire is not connected. Then connect it to CLK8 or MPUCLK; the LED should be somewhat dimmer, indicating that the signal is high part of the

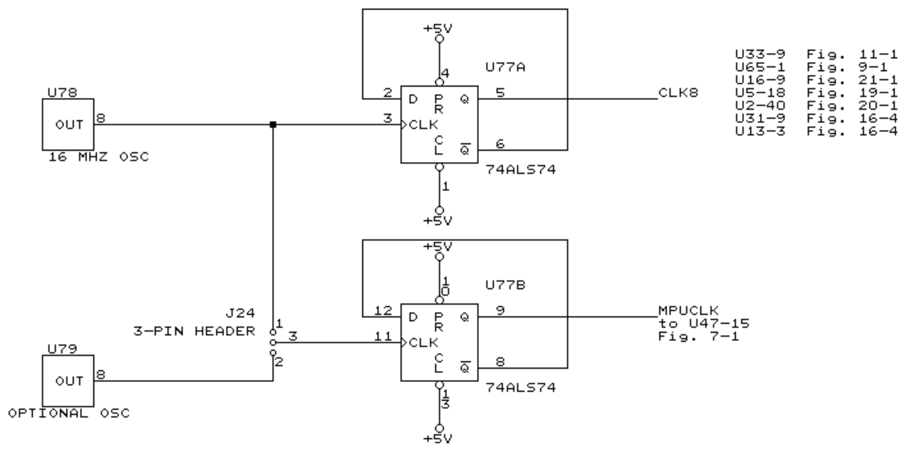


Fig. 5-1. Main clock circuit.

time and low part of the time. Since the LED is flashing on and off so fast you cannot see it, it appears somewhat dimmer than when on continuously.

Next, connect the probe wire to MPUCLK and note its brightness. Then slip off the shorting jumper from J24-1 and note whether the LED gets brighter or darker. Each time you do this, you stop U77b from flipping. Sometimes it will stop in the set state, in which case the LED will be getting a full high voltage and become brighter; other times it will stop in the reset state, in which case the LED will go off. If all this is happening, then all is well.

Chapter 6

68000 Operation

We will do no construction in this chapter; instead, we will take a detailed look at the individual pins of the 68000 and what they do.

6-1. 68000 Pinout

Fig. 6-1 shows some of the wiring to the 68000 microprocessor. Though the 68000 is a 64-pin IC and its wiring looks complex, it is really quite straightforward. Let's go over it pin by pin first.

On the right we see the data bus with its 16 lines labelled D0 through D15, and the address bus with its 23 lines labelled A1 through A23. In case you've noticed that one is missing, you're right - there is no A0. Its function is handled by LDS and UDS.

Let's look at the control lines on the left side more carefully; they are labelled with arrows to indicate whether they are inputs or outputs or, in some cases, both.

At the top left are FC0, FC1, and FC2. These three active-high lines output a Function Code which can be externally decoded to indicate what the 68000 is doing internally; it also could be used to increase the 68000's memory up to 64 megabytes if necessary.

E (an Enable clock), VMA (valid memory address), and VPA (valid peripheral address) are useful if the 68000 is used with older I/O chips, those originally intended for Motorola's 6800 processor. VPA is also provides some interrupt information, and that is the only function the SK68K system will use it for.

IPL0, IPL1, and IPL2 are interrupt level inputs. We will discuss interrupts later; now let us just say that outside events (such as a keyboard) can interrupt whatever the 68000 is doing and cause it to respond. These three inputs tell the 68000 whether an interrupt is being asked for, and what kind of an interrupt it is.

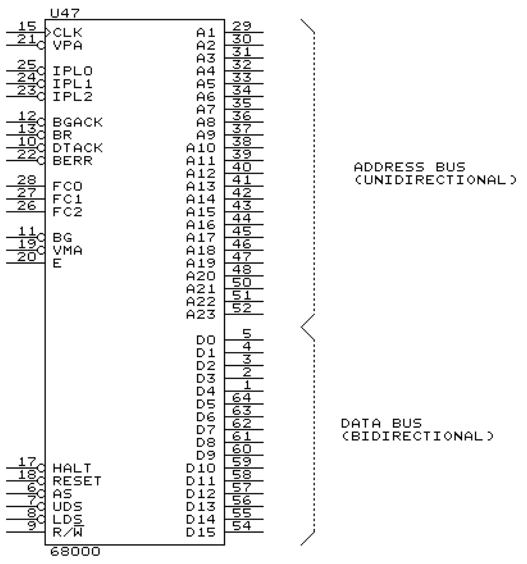


Fig. 6-1. 68000 microprocessor pinout.

The $\overline{\text{RESET}}$ and $\overline{\text{HALT}}$ inputs come from the 555 circuit in Fig. 4-1, but these two pins are also outputs. That explains why an open-collector 7406 inverter was used to drive them; occasionally the 68000 may output a low on one of these two lines, which would conflict with the normally-high output of a standard inverter (such as a 7404) and cause excessive current flow.

$\overline{\text{BR}}$, $\overline{\text{BG}}$, and $\overline{\text{BGACK}}$ are used when DMA circuitry is used. If DMA were used, the DMA controller circuit would send a Bus Request ($\overline{\text{BR}}$) to the 68000, which would release the data and address busses and return a Bus Granted (BG). The DMA controller would then send a Bus Grant Acknowledge (BGACK) signal to confirm that it has control of the busses, and temporarily take over the system while the 68000 sits back and waits.

LDS and UDS replace address line A0 in an interesting way. Since the 68000 has a 16-bit data bus whereas memory is divided into 8-bit bytes, the data bus can access two bytes at a time. The memory is wired so that half of memory - all the odd-numbered locations - connects to the 'lower' part of the data bus (bits D0 through D7), while the other half of memory - all the even-numbered locations - connects to the 'upper' part of the data bus (bits D8 through D15). The 68000 asserts $\overline{\text{LDS}}$ (lower data strobe) when it wants to use the lower half of the data bus, asserts $\overline{\text{UDS}}$ (upper data strobe) if it wants to use the upper half of the data bus, or asserts both if it wants to transfer 16 bits on the entire data bus. Thus an odd address turns on $\overline{\text{LDS}}$ while an even address turns on $\overline{\text{UDS}}$; this is similar to the function of A0, since A0 is 0 for an even address and 1 for an odd address.

$\overline{\text{AS}}$ is an address strobe which is generally asserted by the 68000 at the same time as either $\overline{\text{LDS}}$ or $\overline{\text{UDS}}$, and simply tells external circuitry (mainly address decoders) that there is a valid address on the address bus. This is important, since the address bus often carries data which is not meaningful; there has to be a way to prevent address decoders from responding to it in error.

Winding down the home stretch, we get to R/$\overline{\text{W}}$ which stands for Read/not Write. This is a signal used by the 68000 to tell other circuitry whether it wants to read data in (when R/$\overline{\text{W}}$ is high) or write data out (when R/$\overline{\text{W}}$ is low). Thus R/$\overline{\text{W}}$ would be high when data goes from the RAM or ROM to the 68000, whereas it would be low when data goes from the 68000 to RAM.

$\overline{\text{BERR}}$ is an input to the 68000, used by external circuitry to tell the 68000 that something has gone wrong on one of the busses. We will see how this is done later.

Finally, $\overline{\text{DTACK}}$ stands for Data Transfer Acknowledge. Whenever the 68000 wants to read or write to memory or an I/O device, it (a) puts the address on the address bus, (b) puts a high or low on R/$\overline{\text{W}}$, (c) outputs the address strobe and $\overline{\text{LDS}}$ and/or $\overline{\text{UDS}}$, and then sits back and waits. It waits until it either gets back $\overline{\text{DTACK}}$, indicating that the transfer is finished, or $\overline{\text{BERR}}$, indicating that something went wrong. When $\overline{\text{DTACK}}$ is received, then the 68000 goes on to the next step.

If $\overline{\text{DTACK}}$ were grounded, the 68000 would always assume that the transfer was finished really fast, and would zip along at maximum speed. In most cases, though, $\overline{\text{DTACK}}$ comes from an external timer circuit of some kind which gives memory and I/O just enough time to finish their job. If a certain memory or I/O device is particularly slow, $\overline{\text{DTACK}}$ can be delayed so the 68000 waits for it to finish.

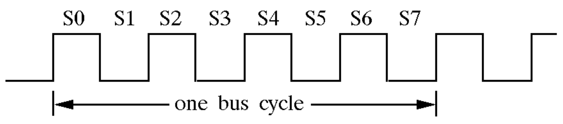
In practice, each 68000 memory or I/O access takes a certain minimum amount of time, measured in cycles of the MPUCLK signal. If $\overline{\text{DTACK}}$ is delayed, even just an instant, the 68000 lets an entire extra clock cycle slip by and checks again. If $\overline{\text{DTACK}}$ is still off, then the 68000 waits another clock cycle, and so on. Each of these extra clock cycles is called a wait state. The ideal case would be to have everything fast enough so that the 68000 can go right on without wait states; some computers have slower memory or I/O and run with one or even more wait states, which obviously slows everything down. (You'll be happy to know that the SK68K runs with no wait states.)

6-2. 68000 TIMING

Note: the rest of this chapter provides more detailed information about the 68000 than you really need to understand its operation at this point. Although we present it here for completeness, we suggest that you skip ahead to the next chapter, and return here to read the rest of this material after you have finished with the rest of this volume.

As described in Chapter 5, the master 68000 clock is called MPUCLK. In a basic SK68K system, this clock could be as slow as 8 MHz or as fast as 12.5 MHz. It is this clock which governs how fast the 68000 performs its operations.

Slightly idealized, MPUCLK looks like this:



Let us assume that MPUCLK runs at 8 MHz. In the simplest case, four complete cycles of the clock constitute a *Bus Cycle* as shown in the figure. At an 8 MHz frequency, one clock cycle is equal to

$$\frac{1}{8 \times 10^6} \text{ second,}$$

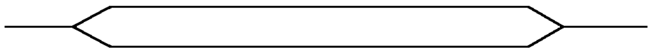
or 125 nanoseconds (ns). Thus a bus cycle is four times that - 500 ns or 1/2 microsecond (µs). (As we will see in a moment, a bus cycle could be longer - this computed time is the minimum.)

The bus cycle is called that because it represents the time required for one complete bus operation. In other words, a read from memory (which involves a bus operation), or a write to memory (which also involves a complete bus operation), requires one bus cycle.

The reason why the clock must run four times faster than a bus cycle is that the 68000 uses MPUCLK edges to time its own internal operations. The above figure shows the four clock cycles divided into eight half-cycles, each one of which has an edge. These half-cycles are called *states*, and are numbered S0 (meaning *state 0*) through S7. Each of these states begins with a clock edge - for example, state S0 begins with a rising edge, S1 begins with a falling edge, S2 again with a rising edge, and so on. These eight edges provide eight different times during a bus cycle at which the 68000 can trigger flip-flops or perform various other operations. For example, if the rising edge at the beginning of S0 causes something to happen, we will say that it happens *at the beginning of state 0* or perhaps that it occurs *during state 0*.

With this in mind, let us look at Fig. 6-2. Here we see the MPUCLK in relation to a bus cycle. Note that this bus cycle is preceded and followed by other bus cycles. Thus there was probably a state 7 just before state 0, and there will be another state 0 just after state 7 of the current bus cycle. The thing to remember is that the 68000 clock never stops - in normal operation, the 68000 just keeps running one bus cycle after another.

Before continuing, let us explain some of the symbols used in Fig. 6-2. The waveform labelled “DATA BUS", for example, looks like this:



Keeping in mind that the data bus consists of 16 lines, it is obviously not practical to show the signal on each and every one of those lines. Instead, we attempt to show all twelve signals on one waveform without being specific as to which lines are high and which are low. The waveform shown starts off with a thin line at the left, which then opens up to one curve which

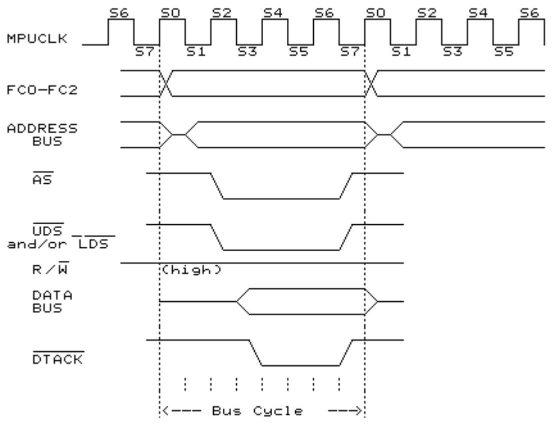


Fig. 6-2. Normal read cycle timing

goes up to a high, and another which goes down to a low. The intent is to represent that "some data bus lines - we will not specify which - are high, and others are low."

The single line at the left and right, which is shown halfway between a high and a low, clearly cannot represent an actual signal; instead, it is supposed to simply tell us that there may be some data on the bus at that time, but we don't care what it is. In a sense, we may think of it as useless garbage or, as some Motorola literature refers to it, *irrelevant data*..

Looking at Fig. 6-2, then, we see that the data bus has irrelevant garbage data for roughly the first half of the bus cycle, some real data for the second half of the cycle, and then goes back to irrelevant data after the bus cycle ends.

We see a similar situation on the address bus, except that here the irrelevant data exists for only a short time near the beginning of the cycle. On the FC0 through FC2 lines, on the other hand, the data changes very quickly from one set of valid data to another, just after the beginning and end of the bus cycle.

With that background, let us see what happens when the 68000 wishes to do a read from memory. To do so, it performs a *read bus cycle*, which is shown in Fig. 6-2. The 68000 then does the following:

Start of S0:	a. Remove whatever address was on the address bus from the previous bus cycle,
	b. Make R/$\overline{W}$ high to indicate it wants to read,
	c. Place the new function code on pins FC0 through FC2.
Start of S1:	Place on the address bus the address of the location it wants to read from.
Start of S2:	a. Assert the address strobe $\overline{AS}$

b. Assert $\overline{\text{UDS}}$ and/or $\overline{\text{LDS}}$. It asserts $\overline{\text{UDS}}$ for a byte read from an even address, LDS for a byte read from an odd address, or both strobes for a 16-bit read from a pair of addresses, one even and the other odd.

The 68000 now waits. The address decoder, which receives $\overline{\text{AS}}$, $\overline{\text{UDS}}$, LDS, and the address on the address bus, sends an appropriate enable signal to the ROM, RAM, or I/O interface (depending on the address). This device is now supposed to do a read and send the desired data to the data bus. At about the same time, it is supposed to assert DTACK to tell the 68000 that the data is available. In order to continue as shown, the 68000 needs DTACK before the end of state 4, while the data itself must be on the data bus before the end of state 7. If all this happens on time, then the 68000 proceeds as follows:

Start of S7: a. The data on the data bus is latched inside the 68000,

b. $\overline{\text{AS}}$, and $\overline{\text{UDS}}$ and/or $\overline{\text{LDS}}$ are negated (they go back high).

Once $\overline{\text{AS}}$, and $\overline{\text{UDS}}$ and/or $\overline{\text{LDS}}$ go off, the memory or I/O device is supposed to turn off DTACK and also remove the data from the bus. This completes the bus cycle, at the end of which the 68000 removes the FC0 through FC2 signals and removes the address from the address bus, in preparation for the next cycle.

As mentioned above, DTACK is supposed to arrive before the end of state 4. What happens if DTACK is delayed (by slow memory, for example)? Fig. 6-3 shows what happens.

If DTACK arrives too late, instead of going from state 4 into state 5, the 68000 pauses and waits for DTACK. Since the MPUCLK is still running,

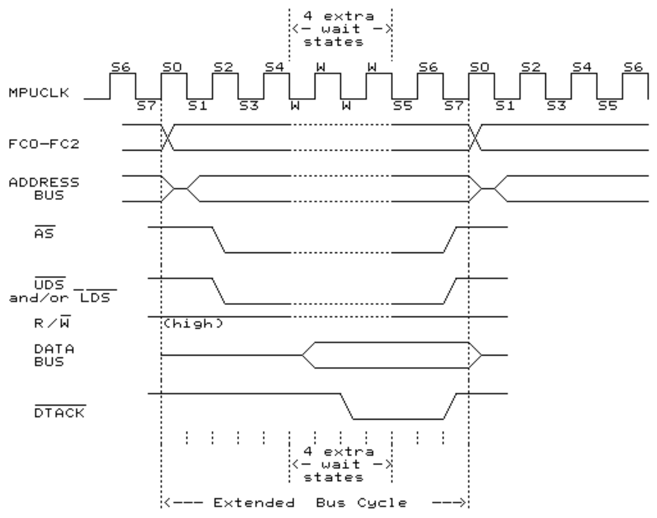

4 extra <- wait -> states
S6 S0 S2 S4 W W S6 S0 S2 S4 S6
MPUCLK
S7 S1 S3 W W S5 S7 S1 S3 S5
FC0-FC2
ADDRESS BUS
AS
UDS and/or LDS
R/W
(high)
DATA BUS
DTACK
4 extra <- wait -> states
<--- Extended Bus Cycle --->

Fig. 6-3. Extended read cycle timing.

extra clock cycles are now inserted between state 4 and state 5; once $\overline{\text{DTACK}}$ arrives, the 68000 will begin state 5 at the next falling edge. Since each clock cycle is two states, this means that even numbers of states (called *wait states*) are inserted into the bus cycle. Fig. 6-3 shows DRAM arriving so late that four wait states (labelled W) are inserted between S4 and S5.

In the actual SK68K computer, the DRAM is fast enough that DTACK comes in time; the ROM and static RAM are slower, on the other hand, and so they operate with two wait states. The largest number of wait states occurs with some types of XT-compatible cards plugged into the expansion connectors; some of the video boards are slow enough that they may insert hundreds of wait states.

This type of operation, where the 68000 waits for memory or I/O devices to finish their operation, is called *asynchronous*. While the term "asynchronous" implies "not synchronized", we see that signals really are very carefully synchronized with MPUCLK after all, so the term may not be entirely accurate. But it does help us differentiate from *synchronous* operation, which is used in almost all earlier microprocessors. In these processors, every bus cycle is exactly the same length. In a synchronous system, the cycle length must be adjusted to allow the microprocessor to keep up with the *slowest* memory or I/O device, which prevents us from taking advantage of faster speeds possible in some parts of the system. The asynchronous operation of the 68000, on the other hand, allows the system to run at the maximum speed possible at any given time. (For use with older-design I/O interfaces, the 68000 can also run in a synchronous mode. This mode is not, however, used in the SK68K.)

Fig. 6-4 shows bus operation during a write to memory or I/O. The waveforms are very similar to those of a read cycle, except for three differences:

1. R/$\overline{\text{W}}$ now goes low during the cycle, to tell memory or I/O devices that a write is occurring instead of a read.

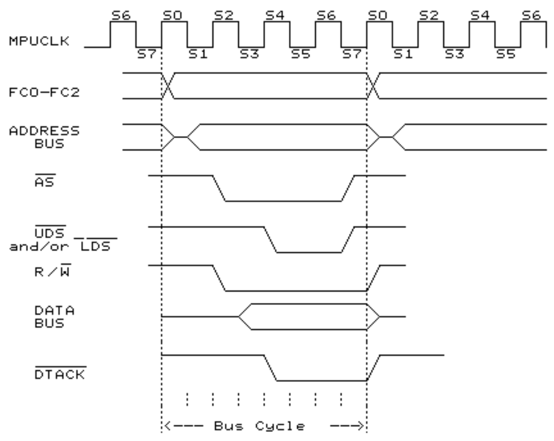


Fig. 6-4. Normal write cycle timing.

2. $\overline{\text{UDS}}$ and $\overline{\text{LDS}}$ are delayed, so they do not start until S4.
3. Data on the data bus now comes from the 68000, instead of coming from memory or an I/O interface. It appears on the data bus at the beginning of S3, and should be latched off the bus by the destination device at the end of $\overline{\text{LDS}}$ or $\overline{\text{UDS}}$.

So far, we have described the read and write bus cycles separately. As we have seen, the minimum such cycles are 4 clock periods or 8 cycles, although they can be stretched longer by wait cycles. In most cases, as the computer runs it will have many more read cycles than write cycles, but these cycles will continuously follow each other unless the computer is somehow halted.

There is one particular instruction, though, that has a minimum bus cycle of 20 states, shown in Fig. 6-5. This is the TAS or "Test and Set" instruction, which is generally used only when the 68000 is being used for multi-tasking. In these applications, the operating system software generally uses a software *flag* (a bit in memory) to indicate whether the system is free to start another program, or whether it is busy. If the operating system software wants to run a particular program, it first checks this flag to see whether the system is busy. If not, then it sets the flag (to indicate that the system is now busy) and then starts the desired program. It is important to be able to test the flag and then immediately set it in the same instruction, because this prevents two processes testing the flag at almost the same time, and both starting up under the impression that the system is free.

The TAS instruction therefore contains a 20-state bus cycle which consists of reading the flag from memory into the 68000, four states during which the 68000 can modify the data just read, and then another set of states to write the new flag data back into memory. Whether you think of this *read-modify-write* operation as one big cycle, or as two smaller cycles sepa-

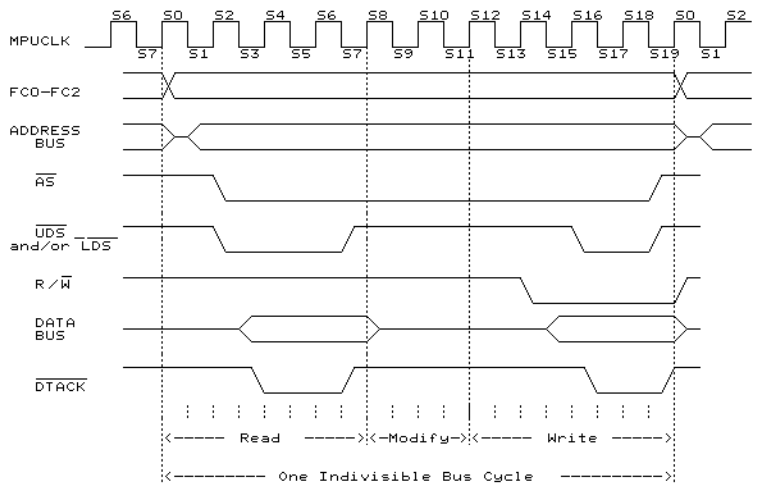

MPUCLK
S6 S0 S2 S4 S6 S8 S10 S12 S14 S16 S18 S0 S2
S7 S1 S3 S5 S7 S9 S11 S13 S15 S17 S19 S1
FC0–FC2
ADDRESS BUS
AS
UDS and/or LDS
R/W
DATA BUS
DTACK
<----- Read ----->
<-Modify->
<----- Write ----->
<--------- One Indivisible Bus Cycle --------->

Fig. 6-5. Read-modify-write cycle timing.

rated by a modify, the fact is that they cannot be separated. The TAS instruction is used very seldom; in fact, the circuitry of the SK68K computer does not support it.

We have omitted specific details of bus timing in the foregoing description, giving only a rough idea of the sequence in which things happen. These specific timing details can be found in the Motorola 68000 data book. Instead, we will discuss some more general principles of bus operation.

6-3. The Function Code Outputs

As Figs. 6-2 through 6-5 show, the FC0 through FC2 outputs from the 68000 provide a three-bit output during the entire bus cycle (except for a slight delay right at the beginning of S0). The following table shows the meaning of these three bits.

FC2	FC1	FC0	Meaning
0	0	0	Not used
0	0	1	User data
0	1	0	User program
0	1	1	Not used
1	0	0	Not used
1	0	1	Supervisor data
1	1	0	Supervisor program
1	1	1	Interrupt Acknowledge

These outputs can therefore inform external circuitry what is happening inside the 68000. They could, for example, be used to switch in different banks of memory, so that programs and data could be stored in different memory.

6-4. Interrupt Operation

As shown in the above table, when all three FC outputs are a 1, the 68000 is signalling an Interrupt Acknowledge. As shown in Fig. 6-6, this is used to generate the VPA signal which tells the 68000 how to process an interrupt request.

The interrupt system in a computer is used to allow external devices, such as I/O interfaces, to interrupt a running program. While the program is interrupted, the microprocessor can execute a different program called an *interrupt service routine* (ISR), which can in some way service the I/O device. When the ISR is finished, the interrupted program continues from the point where it was stopped as though nothing had happened.

The interrupt system in a 68000 is quite extensive. In addition to being caused by external hardware, 68000 interrupts can also be caused by so-called *trap* instructions, and by certain kinds of programming errors (such as using invalid operation codes or trying to divide by zero.) These

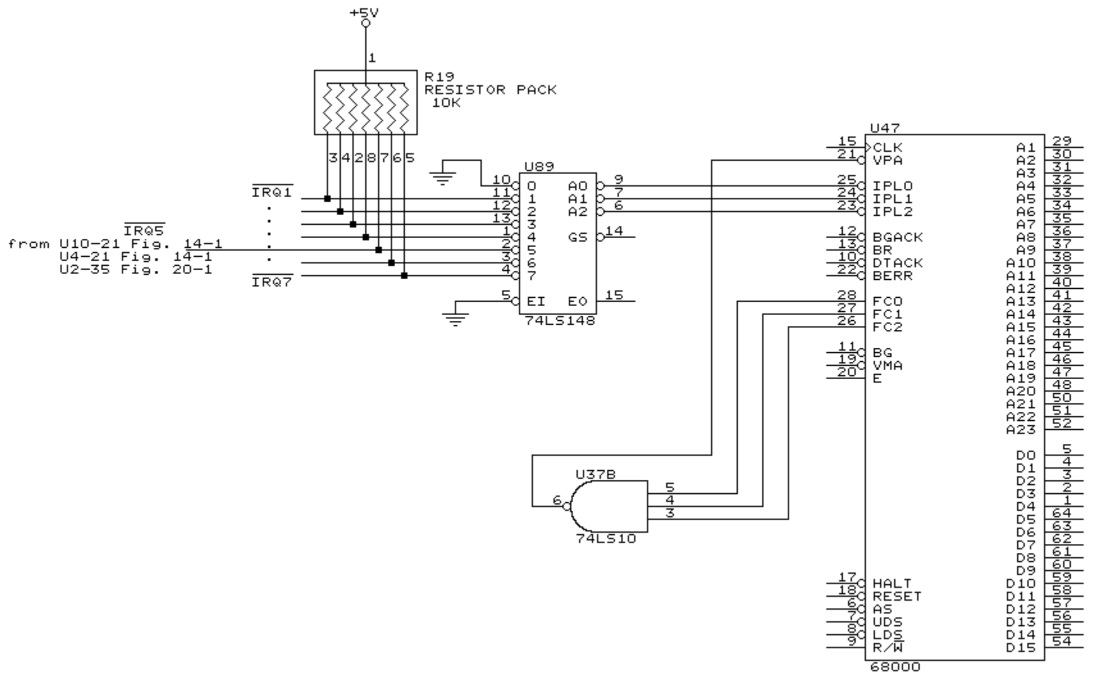


Fig. 6-6. 68000 Interrupt circuitry.

interrupt sources fall into the category of software, and so we will not look at them further at this time. But we will look at hardware interrupts.

Fig. 6-6 shows the circuitry involved (we will not actually install it until the next chapter). In normal operation, the seven IRQn inputs into U89 are all pulled high by the seven 10K resistors, and thus U89 outputs three high levels to the IPL inputs of the 68000. The 68000 ignores this condition and executes its normal program.

But suppose that a character has been received by the DUART, and the DUART tries to signal the 68000 to accept that character. Assuming that the DUART has been properly programmed to do this, it will output a low signal on the IRQ5 input to U89.

U89 is a *priority encoder*. If it receives a low on one of its inputs, U89 outputs the number of that input on its A2 - A0 outputs. For example, when IRQ5 is low, it outputs the number 101, a binary 5. This number is called the *interrupt level*. (Noting that the A outputs as well as the IPL inputs are active low, the 101 bit pattern is actually represented as low-high-low.) The word *priority* in the name of U89 becomes important here - it means that if *several* of U89’s inputs are low, U89 outputs the number of the highest low input. For example, if IRQ6, IRQ5, and IRQ2 are all low, then U89 would output the number 6.

When the 68000 receives the number 101, it checks its status register to see whether such an interrupt is currently allowed. If not, then the 101 is temporarily ignored. If it is allowed, then the 68000 completes the current instruction and then begins *exception processing*.

Exception Processing is the generalized name for processing an interrupt on the 68000 (either hardware or software). The 68000 begins by stopping the current program, switching to supervisor mode (if not already in it), and then outputting a 111 on the FC outputs to acknowledge the interrupt. At this point, it needs to know where to find the ISR which is supposed to process the interrupt.

There is room in low memory, between locations $0 and $3FF, for up to 256 *exception vectors*, which are pointers to interrupt service routines. These vectors would normally be placed there by some program, though they could also be in ROM. When the 68000 outputs the 111 on the FC outputs, it also outputs the number of the interrupt on the A2 - A0 address lines. One of two things can now happen in a typical system:

1. The interrupting device can tell the 68000 which vector to use to find the ISR. It would do this by placing a high on the VPA line, and putting the vector number (an eight-bit number between 0 and 255) on data bus lines D7 through D0.

or

2. External circuitry can place a low on the VPA line, in which case the 68000 will automatically choose one of seven vectors, depending on the interrupt level. Since the 68000 chooses its own interrupt vector, this is called *auto-vectoring*.

As shown in Fig. 6-6, U37b automatically sends a low to the VPA pin of the 68000 as soon as it receives three highs on the FC lines, and so the SK68K uses auto-vectoring to choose a hardware interrupt vector.

Chapter 7

68000 Operation In An Open Loop

Now that we have functioning clock and reset circuits, and are familiar with some of the pins on the 68000, it is time to get it to do something. Normally, you need quite a bit of external hardware to get any microprocessor running, but there is a way of fooling the 68000 into thinking there is an entire computer connected even though there is almost nothing there.

7-1. Discussion

Fig. 7-1 shows how we are going to get the 68000 to work with the minimum of external hardware.

In order to minimize the amount of extra wiring we have to do, we will take advantage of circuitry already on the printed circuit board which will be needed later anyway. For example, the RESET, HALT, and CLK signals are already connected on the printed circuit board, and we don't need to do anything to them at all.

U37b is a NAND gate which generates a low $\overline{\text{VPA}}$ when FC0, FC1, and FC2 are all high, but keeps $\overline{\text{VPA}}$ high at all other times. As explained in Chapter 6, it is used during interrupt processing to tell the 68000 to use auto-vectoring; for now, it will simply keep $\overline{\text{VPA}}$ high.

U89 is the priority encoder IC, also used only during interrupts. For now, however, the seven resistors in R19 are keeping all of the $\overline{\text{IRQ}}$ lines high; since they are active low, this negates them. U89 therefore sees no interrupt request, and so it sends the number 000 to the 68000, telling it there are no interrupt requests. We could have achieved the same thing by grounding the three $\overline{\text{IPL}}$ pins of the 68000 and connecting $\overline{\text{VPA}}$ high, but we can save ourselves that job by installing R19, U89, and U37, which we would have had to do eventually anyway.

Two of the 68000's inputs are already connected to a high to negate $\overline{\text{BR}}$ and $\overline{\text{BGACK}}$, and several pins (including the entire address bus) are la-

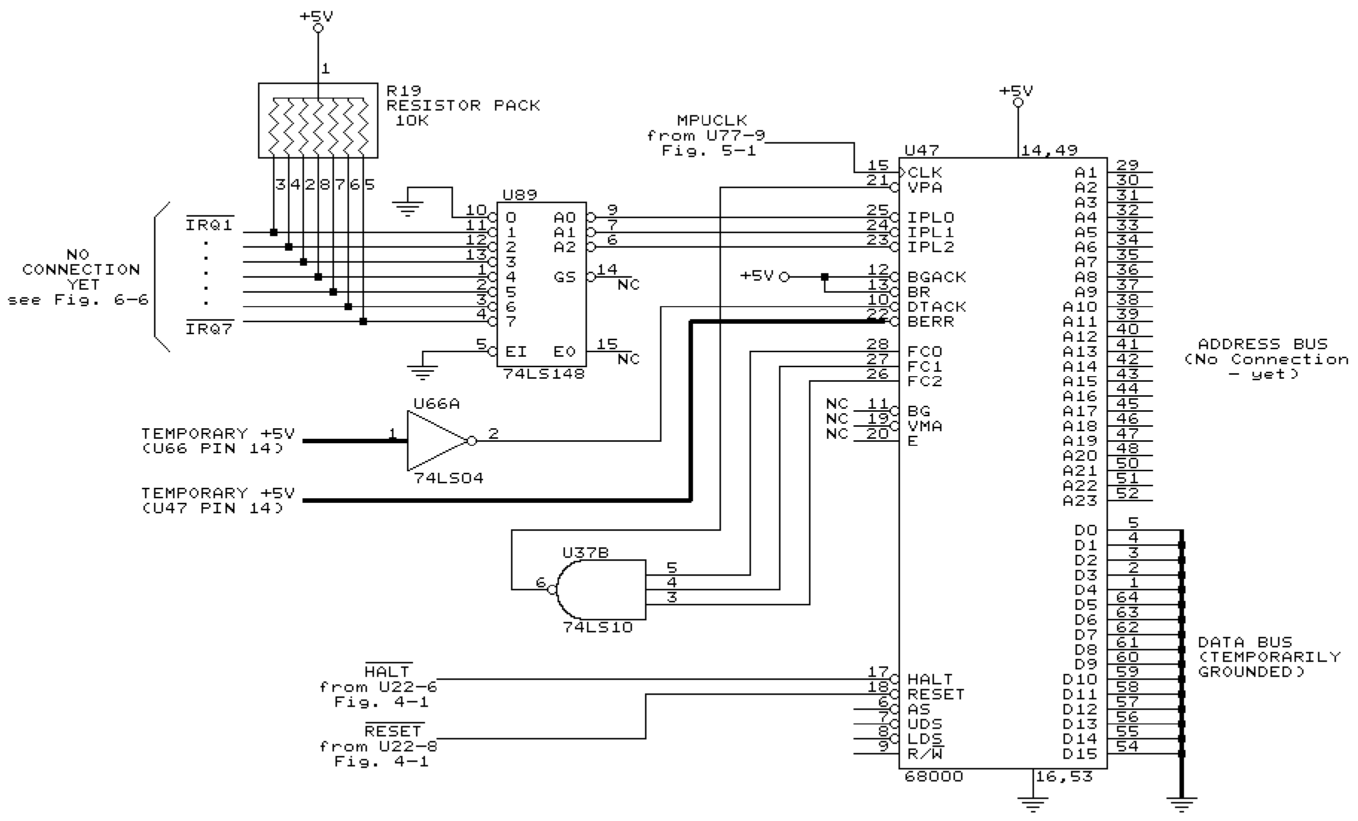

+5V
R19
RESISTOR PACK
10K
NO CONNECTION YET see Fig. 6-6
IRQ1
IRQ7
U89
74LS148
EI
EO
GS
A0
A1
A2
NC
MPUCLK from U77-9 Fig. 5-1
U47
68000
+5V
CLK
VPA
IPL0
IPL1
IPL2
BGACK
BR
DTACK
BERR
FC0
FC1
FC2
BG
VMA
E
HALT
RESET
AS
UDS
LDS
R/W
U66A
74LS04
TEMPORARY +5V (U66 PIN 14)
TEMPORARY +5V (U47 PIN 14)
U37B
74LS10
HALT from U22-6 Fig. 4-1
RESET from U22-8 Fig. 4-1
ADDRESS BUS (No Connection - yet)
DATA BUS (TEMPORARILY GROUNDED)
14,49
16,53

Fig. 7-1. Test circuit for the 68000.

belled NC to show that they do not, as yet, go anywhere. We need do nothing with these as yet.

That leaves the three sets of connections shown in heavy black on Fig. 7-1. First, $\overline{\text{BERR}}$ must be held high (negated) so the 68000 doesn't think a bus error has occurred; this is easily done by installing a short jumper between pins 14 and 22 of the 68000, since pin 14 is at +5 volts.

Although $\overline{\text{DTACK}}$ will normally carry a meaningful signal, for now we want to ground it to make the 68000 think that all is well on the outside. But since inverter U66 was already previously installed, we instead connect U66a input to +5 volts with a jumper from U66 pin 1 to pin 14.

Finally, when the 68000 is running, it wants to keep fetching instructions and addresses out of memory; we have to provide it with some meaningful data so it can keep going. We do so by grounding all 16 lines of the data bus. In this way, every time the 68000 tries to read anything out of memory, it will get back the number 0000. As it turns out, there is a a valid 68000 machine language instruction which says "OR a 0 to register D0", abbreviated OR.B #0,D0, which consists of four bytes of 00.

Though this OR instruction does absolutely nothing, the 68000 thinks all 16 megabytes of its memory are full of nothing but 4 million OR instructions, and so it starts doing them one after another. When it gets to the top of memory at $FFFFFF, it simply starts all over again from $000000.

7-2. Construction

Now let's wire up the required components and see what happens. Install the following:

	64 pin socket for U47 (but don't plug in the 68000 yet)
U37	74LS10 triple 3-input NAND, and its socket
U89	74LS148 priority encoder, and its socket
R19	10K single-in-line package. Its pin 1, identified by a white line, points toward J25
C14 and C48	0.1 µF disc capacitors
	A short wire jumper from U47-14 to U47-22 to negate BERR.
	A short wire jumper from U66-1 to U66-14 to assert DTACK. Both of these jumpers will be removed later, so do it neatly and in a way which is easily unsoldered later.

Grounding the entire data bus can be messy, so we will do it another way. The data bus is connected to the two EPROM sockets (U20 and U27) and the two static RAM sockets (U21 and U28) as shown in Fig. 7-2. Start by installing a 28-pin socket at U27 and a 24-pin socket at U21.

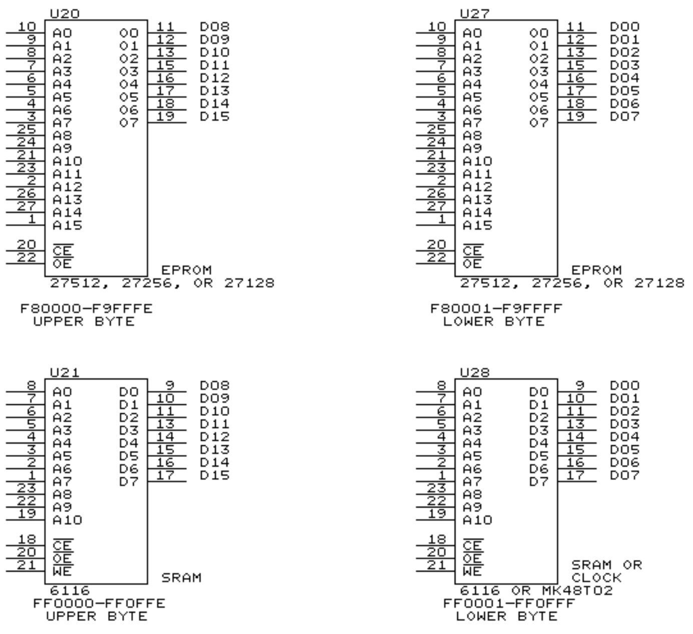


Fig. 7-2. EPROM and SRAM data bus wiring.

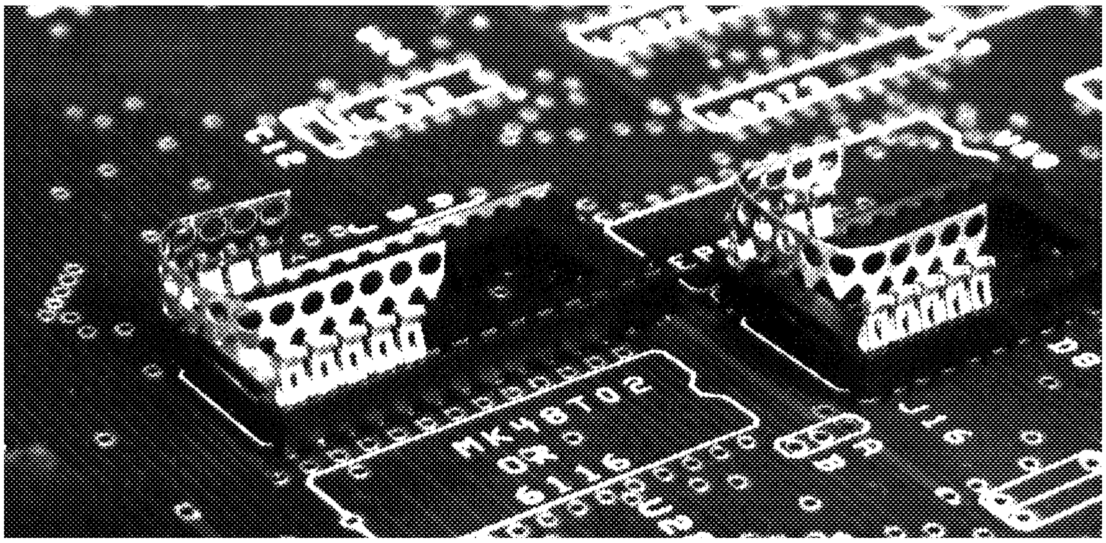


Fig. 7-3. Grounding the data bus with Molex pins.

On U27, data bus bits D0 through D7 go to pins 11-13 and pins 15-19, while pin 14 (not shown) is conveniently grounded. Thus we need to short together pins 11 through 19. In the same way, on U21, data bus bits D8 through D15 go to pins 9-11 and pins 13-17, while pin 12 (not shown) is conveniently grounded. Thus we need to short together pins 9 through 17. On both ICs, these are the bottom four pins on the left and the bottom five pins on the right. Instead of soldering, we take a strip of Molex Soldercon pins and insert them into the sockets as shown in Fig. 7-3, with four pins on the left, five pins on the right, and a section of six or so pins bent in a U shape down below. These Molex pins are normally sold as an inexpensive substitute for sockets; they consist of individual clips joined by a perforated carrier strip which would normally be broken off after soldering. In our case, we insert the thin pins (which would normally be soldered to a board) into the socket, and use the entire strip as one big short circuit.

7-3. Testing

Finally, plug in the 68000 (be careful not to bend pins) and turn on the power. Connect your LED probe to pin 52 of the 68000, which is address line A23; if all is well, the LED will light for about 2 seconds, go off for about 2 seconds, and so on. It may operate faster if your SK68K clock is faster than 8 MHz.

If the LED does not flash at the 2 seconds on, 2 seconds off rate, recheck the signal at every pin of the 68000. Look especially for the low on $\overline{\text{DTACK}}$, lows on all data lines, a high on $\overline{\text{BERR}}$, $\overline{\text{HALT}}$, and $\overline{\text{RESET}}$, and the clock signal (LED slightly less than full brightness) on CLK.

If the LED flashes as expected, all is probably well. What's happening is that the 68000 is racing through memory (or what it thinks is memory, all 16 megabytes worth), executing OR instructions at its maximum speed, one instruction every microsecond. One complete run through 4 million instructions (4,194,304 instructions, to be exact, one quarter of 16 megabytes or 16,777,216) therefore takes a bit over 4 seconds.

During that time, the address bus is counting off the addresses where the 68000 thinks those instructions are coming from. If you look at some small binary numbers such as 000, 001, 010, 011, 100, 101, 110, 111 etc., you see that the right-most bit alternates 0, 1, 0, 1, 0, 1, ... The second bit also alternates, but slower: 0, 0, 1, 1, 0, 0, ...; the third bit is slower still: 0, 0, 0, 0, 1, 1, 1, 1, ..., and so on. Exactly the same thing occurs on the address bus: A1 alternates back and forth very rapidly, A2 goes half as fast as A1, A3 is half as fast as A2, and so on, all the way up to A23 which alternates from 0 to 1 and back to 0 once every four seconds.

Fig. 7-4 shows some of the waveforms on the upper bits of the address bus. When the LED probe is connected to A23, it flashes on and off once every four seconds. If it is connected to A22, it repeats once every two seconds; on A21 it repeats once per second. As we go down to A20, A19, and so on, it flashes faster and faster, until at A16 (pin 44) it flashes so fast (about 32 times per second) that we can barely see it flicker; A15, flashing about 64 times per second, looks absolutely steady (though not at full brightness).

As a final check, if you have access to an oscilloscope or frequency counter, check each address line to make sure that its frequency is half that of its higher neighbor. This makes sure that there are no accidental shorts between adjacent lines of the address bus. (If using a frequency counter, keep in mind that the actual waveforms are not nearly as precise as the idealized ones of Fig. 7-4; some counters may have difficulty properly counting the frequency of such a square wave.)

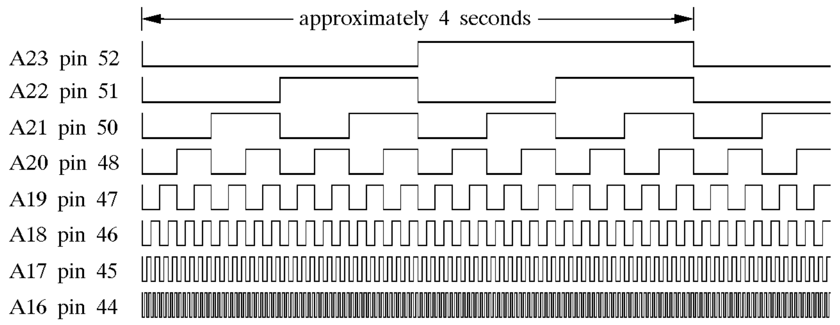


Fig. 7-4. Address bus waveforms during testing.

Chapter 8

The $\overline{\text{MAP}}$ Circuit

In the preceding chapter, we described the connections to the 68000 microprocessor and actually got it to the point where it ran. It is now time to add some of the circuits which make it into a working system.

8-1. Discussion

(Let us begin with a short Detour): We have by now learned that signals could be either active high or active low, and that the name of a signal was often a dead giveaway if it was "notted"; that is, if it had the "not bar" over it. A signal like $\overline{\text{DTACK}}$ was therefore active low because of being notted, whereas FC0 was active high because it did not have a not bar.

Another way to mark a signal in a diagram is by using a *bubble,* which is simply a small circle at the end of a wire. Many engineers and technicians tend to get a bit careless with bubbles so they can't always be trusted, but when properly used they can be very helpful. Look at Fig. 8-1 for an example; this is just a repeat of part of the HALT LED circuitry discussed in Chapter 3. The signal arriving from the left is $\overline{\text{HALT}}$; it has a not bar over the name (we say it is *notted*) to indicate that this signal is low when asserted. In other words, this signal goes low when the 68000 is halted. This is also shown by the bubble at pin 13, the input side of inverter U32f. The output on pin 12, as well as the input into pin 9, has no such bubble and so

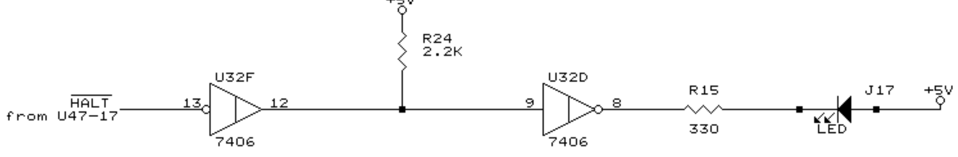


Fig. 8-1. The HALT LED circuit.

we know that this wire is high when the 68000 is halted. Pin 8, the output of U32d, on the other hand, has a bubble again, so we again see that this point is low during a halt. Put another way, pins 13 and 8 are active low, whereas pins 12 and 9 are active high. Here is a prime example where active low and active high circuits are separated by just a tenth of an inch.

It’s important to understand that U32f and U32d are really the same type of device even though their symbols are different. They are both part of U32, a 7406 IC, and are both exactly the same. The fact that one has a bubble on the input whereas the other has it on the output is just a convenient way of drawing the same physical circuit. It is drawn that way to explain what is happening in *this particular circuit*.

Fig. 8-2 shows some more examples of this. The left diagram at (a) is normally called an AND gate; described in words, its job is to “make the output high if input A is high AND input B is high." Thus if both inputs are high, then the output is high. But there is another way of looking at it: if either input is low, then the output is low. Thus we could say “make the output low if A is low OR B is low." That sounds more like an OR gate which works with lows. In a way, the right diagram at (b) is supposed to tell us just that - think of each bubble as being the word ’low’.

Hence if you have two active high circuits and want to AND them to assert an active high output if both inputs are asserted (which means high), then you use the AND gate symbol at the left. But if you have two active low circuits and want to assert an active low output when either input is asserted (which, in this case, means low), then you use the strange-looking OR symbol at the right.

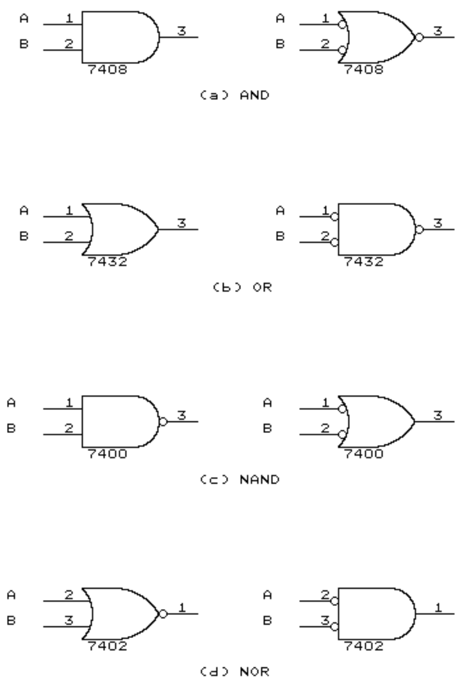


Fig. 8-2. Logic gate equivalents.

It’s important to understand that both symbols in Fig. 8-2 (a) are really the same device - they are just different on paper because they stress a different function. A designer might use either symbol for the same device, depending on what *that particular circuit* is supposed to do. An IC manual, for example, describes a 7408 as a “quadruple 2-input positive-AND gate". This means the 7408 has four 2-input gates which act as “positive-AND". Remembering that ’positive logic’ is just another phrase for ’active high’, this says that the 7408 is used for ANDing in an active high circuit. But the 7408 could just as well be called a “negative OR" (though most IC manuals assume the reader already knows that and hence don’t bother to use those words.)

In the same way, Fig. 8-2 (b) shows a “positive-OR" at the left, and a “negative AND" at the right. In reality, both of these could be used for the same 7432 IC. The left circuit says that “if A is high OR B is high, then the output is high", whereas the right circuit says that “if A is low AND B is low, then the output is low." Both of these sentences say the same thing.

If you think of the bubble as being the word “low", whereas the lack of a bubble means “high", then the left circuit in Fig. 8-2 (c) means "if A is high AND B is high, then the output is low", whereas the right circuit says "if A is low OR B is low, then the output is high." In reality, this is again saying the same thing. Both of these symbols could apply to the same 7400 NAND gate even though the right symbol is really doing an ORing function - if you assume that the inputs are active low, then if either input is asserted low, the output is asserted high.

Likewise, the left circuit in Fig. 8-2 (d) means "if A is high OR B is high, then the output is low", whereas the right circuit says "if A is low AND B is low, then the output is high." Again, this is really saying the same thing. Both of these circuits could apply to the same 7402 NOR gate.

All of this may sound strange to you, but as we go on, you will see that these new symbols greatly help to explain how some parts of the SK68K computer work.

(End of Detour, and back to the $\overline{\text{MAP}}$ circuit.)

When the 68000 first starts operating after it is turned on, it has to know (a) where to place its stack, and (b) where to find the very first instruction to perform. It looks for these two addresses in the first eight bytes of memory, starting at address 0.

In most 68000 computers, however, address 0 is the beginning of RAM; moreover, the RAM is empty when the computer is first turned on. How do these two addresses get there then? The solution is usually to doctor up the address decoder so that at the very beginning it is the ROM, not the RAM, that is at location 0. The ROM is there just long enough for the 68000 to get its two addresses, after which the address decoder removes the ROM and replaces it with RAM. Such a ROM is sometimes called a *phantom*. The MAP circuit of Fig. 8-3 tells the address decoder when to switch in RAM or ROM at address 0.

U90 is an eight-stage shift register - a group of eight flip-flops connected so an input applied to the A and B pins shifts through the register, one flip-flop at a time, as it receives clock pulses. When the computer first starts, or each time it is reset, all the flip-flops in the register are cleared by the $\overline{\text{RESET}}$signal. The QD output, which is normally jumpered out through J25,

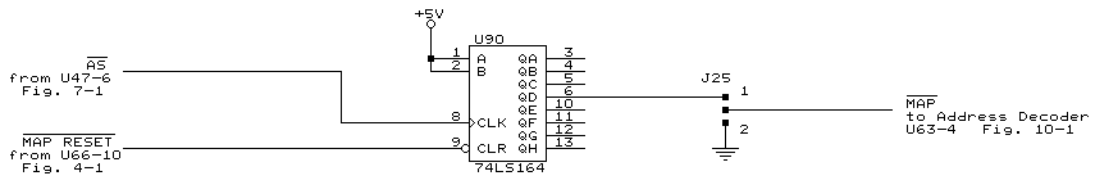


Fig. 8-3. Circuit to generate MAP signal.

then provides a low $\overline{\text{MAP}}$ signal which tells the address decoder to put the ROM at address 0. (In normal operation, there would be a jumper from the center pin of J25 to pin 1.)

When the 68000 starts, it fetches the stack address and program start address from ROM in four 16-bit reads; each one of these reads is accompanied by an AS address strobe. At the same time, the positive edge (i.e., the trailing edge) of each $\overline{\text{AS}}$ strobe clocks U90. Since the A and B shift register inputs are connected to a high (+5 volts), each clock pulse shifts a high into the register. After the first clock pulse, that high gets to QA; after the second pulse it gets to QB, and so on. After exactly four $\overline{\text{AS}}$ strobes, that high gets to QD, out the $\overline{\text{MAP}}$lead to the address decoder, and tells the decoder to disconnect the ROM from address 0 and substitute RAM instead of ROM.

In normal operation, therefore, when the computer first starts, the $\overline{\text{MAP}}$ signal is low and therefore the 68000 sees ROM at address 0. After four $\overline{\text{AS}}$ pulses - exactly long enough for the 68000 to read eight bytes of data from the ROM - $\overline{\text{MAP}}$ goes high and address 0 becomes RAM instead of ROM.

In the SK68K, however, the RAM used at address 0 is dynamic RAM. If we want to use the computer without the DRAM, then we must disable the MAP circuit and keep ROM at address 0. This can be done by jumpering the center pin of J25 to position 2, which keeps MAP low at all times.

8-2. Construction

Reminder:

There should still be two jumpers on your board, left over from Chapter 7: one connecting U47 pins 14 and 22 to negate the 68000's BERR, signal, and one connecting U66 pins 1 and 14 to assert the 68000's $\overline{\text{DTACK}}$ signal. You should also still have the two sets of Molex pins plugged into the U21 and U27 sockets to provide zeroes on the data bus. Leave these until we tell you to remove them.

Now install the following:

U90	74LS164 and its socket

J25	three-pin header. Place a shorting plug from the center pin to position 1.

U66 is already installed from a previous step.

8-3. Testing

Turn on the power and verify that the $\overline{\text{MAP}}$ signal on the center pin of J25 goes low while you short the reset pins (J23), and goes high about a second later, at the same time as the HALT LED goes off. (Unless you have a good quality oscilloscope, it is very difficult to verify that exactly four $\overline{\text{AS}}$ pulses go by before $\overline{\text{MAP}}$ becomes high, so we will have to take it on faith.)

Chapter 9

The Bus Error Circuit

The Bus Error circuit is somewhat unique to the 68000. It acts much like automobile insurance - It is there in case of problem, but hopefully you will not need it.

9-1. Discussion

Each time the 68000 wants to access memory or I/O, it sends out the address, LDS and/or UDS, R/W, and AS. Then it sits back and waits for the external circuitry to respond. If all goes well, the external circuits are supposed to return a low on DTACK (data transfer acknowledge); if something goes wrong, they are supposed to return a low on BERR (bus error).

The DTACK signal is supposed to tell the 68000 how fast it can go; slow memory or I/O tells the 68000 to slow down and insert wait states. But suppose the 68000 program accidentally calls some address at which there is no memory or I/O - what then? Since there is nothing to generate a DTACK signal, the 68000 might sit there forever, waiting. That's where the bus error circuit comes in - its job is to detect the lack of DTACK after some time and generate BERR, which then sends the 68000 into an error recovery procedure.

When we first powered up the 68000 in Chapter 7, we forced DTACK to be low and forced BERR to be high. This forced the processor to go at its maximum speed, and made sure that a bus error would never occur. Since we are still providing a fake DTACK, the 68000 will go ahead even though there is no memory in the system yet, so we might as well put in the correct BERR circuit at this time.

Though the circuit in Fig. 9-1 looks complex, actually it consists of just two parts - U65, a 74LS390 dual decade counter, and U76, a 74LS175 quad D flip-flop.

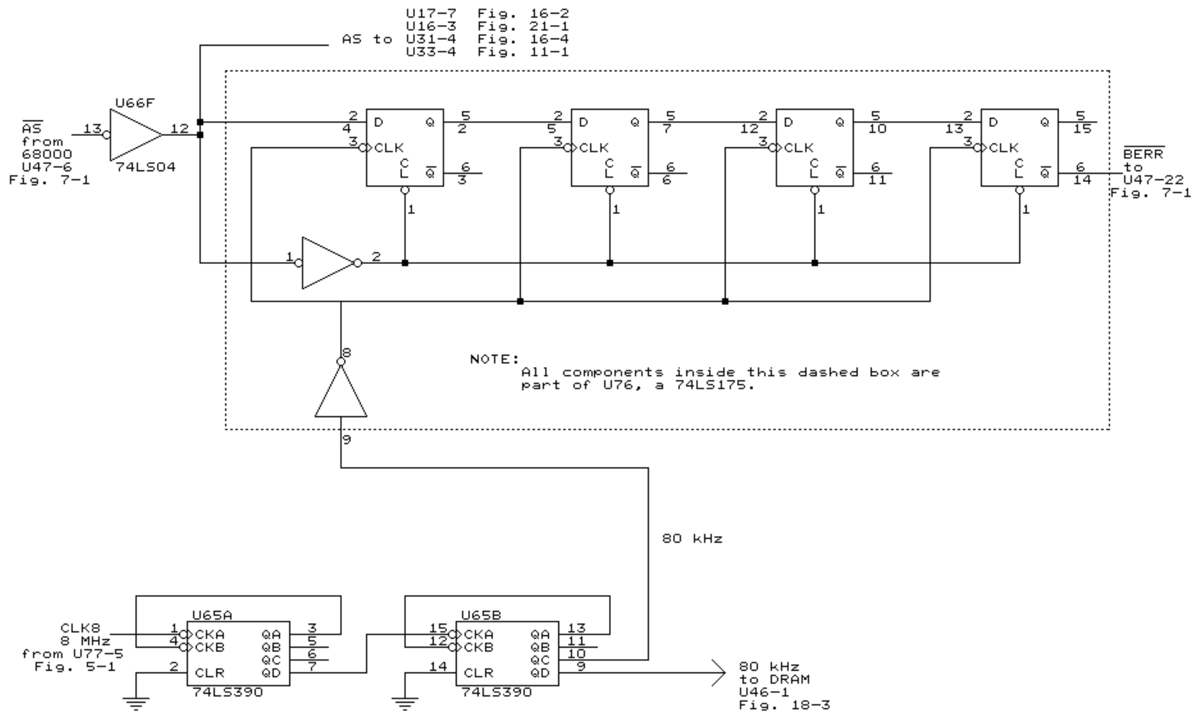

U17-7 Fig. 16-2
U16-3 Fig. 21-1
AS to U31-4 Fig. 16-4
U33-4 Fig. 11-1
AS from 68000 U47-6 Fig. 7-1
U66F
74LS04
BERR to U47-22 Fig. 7-1
NOTE: All components inside this dashed box are part of U76, a 74LS175.
80 kHz
CLK8 8 MHz from U77-5 Fig. 5-1
U65A
74LS390
U65B
74LS390
80 kHz to DRAM U46-1 Fig. 18-3

Fig. 9-1. Circuit to generate BERR.

U65 consists of two decade counters, each of which is wired to divide its input signal by 10. U65a divides the 8 MHz CLK8 signal down to 80 kHz, and U65B further divides that down to 80 kHz. The four flip-flops of U76 are wired as a four-stage shift register clocked by 80 kHz signal from U65b.

The input data comes from the AS address strobe, which is also connected to the clear pin. Before the 68000 starts a memory or I/O access, AS is negated (high), which puts a low on the CLR input and clears all four flip-flops. This makes sure to send a high out the last flip-flop to BERR at the beginning of a memory or I/O access and between accesses.

Once the 68000 starts a memory or I/O access, it asserts AS. Since AS is inverted to AS by U66f, this removes the low on the clear pin, and also sends a high to the D input of the first flip-flop. Each time an 80 kHz clock pulse arrives, this high is shifted one flip-flop right in the register. If the AS signal lasts long enough for four pulses of the 80 kHz clock (about 50 microseconds), then the last flip-flop will set and BERR will go low to tell the 68000 that too much time has passed since the memory or I/O access started.

9-2. Construction

Building the circuit is easy - **remove the jumper** between U47 pins 14 and 22 (which kept BERR negated) and then install

U65	74LS390 dual decade counter and its socket

U76	74LS175 quad D flip-flop and its socket.

(U66f has been installed previously.)

9-2a. Additional Construction Step

If you have a very early SK68K printed circuit board, then your circuit board contains an error, for which we apologize. On your board, the clock input to pin 9 of U76 (shown in Fig. 9-1) goes to the E output from the 68000, instead of going to U65 as shown in the diagram. This earlier connection works well with older XT-compatible boards, but does not work with some of the newer video boards.

To make the change, please cut the trace from U76-9 to U47-20, and install a jumper from U76-9 to U65-10.

9-3. Testing

Using the LED probe, look at pins 4 and 9 of U76. Since $\overline{AS}$ and the 80 kHz clock pulses are both pulses, the LED will glow but only dimly. Then look at pin 2, the Q output of the first flip-flop. Since $\overline{AS}$ and the 80 kHz signal are not in any particular phase relationship - one is at 1 MHz while the other runs at 80 kHz - the first flip-flop will trigger once every few $\overline{AS}$ cycles, but never stay on very long. Thus the flip-flop is mostly off, and the LED should be quite dim. In normal operation, the other flip- flops never get a chance to set, and so the LED should stay dark when testing any of the other Q outputs. $\overline{BERR}$, of course, is a high all the time, so the LED should be bright when testing pin 14.

Chapter 10

The Address Decoder

The address decoder's job is to continuously monitor the high order bits of the address bus, and signal the ROM, RAM, and I/O interfaces whenever an address comes along which is intended for them. As such, it has an extremely important function in keeping the computer operating properly.

10-1. Discussion

The SK68K computer has a single address decoder circuit, but it consists of three ICs. Let's look at Fig. 10-1 to understand how it works.

The heart of the decoder is U63, a 16L8 PAL or Programmable Array Logic element. A PAL is somewhat like a fast ROM - it has a number of input lines and output lines; when a combination of ones and zeroes is presented on the input lines, the PAL looks up in its internal memory what to send out on the output lines. But there are several significant differences between a ROM and a PAL - the ROM is more complex because it has a larger number of possible output combinations; being simpler, the PAL can be made faster. The ROM is meant to store numbers; the PAL is meant to replace logic ICs. In this case, the PAL replaces almost a dozen gates and inverters at a saving of space and money.

The PAL uses its inputs like this:

1. If $\overline{\text{AS}}$ is negated (high), the PAL does nothing; it needs a low $\overline{\text{AS}}$ to ensure that a valid address exists.
2. If address lines A20-A23 are low, representing addresses starting with the binary bits 0000 (i.e., hex addresses from \$000000 through \$0FFFFF, corresponding to a megabyte of dynamic RAM), the $\overline{\text{DRAM}}$ signal is asserted (low). In addition, if the $\overline{\text{CAS}}$input is low, indicating that the dynamic RAM circuitry wants to access a column of dynamic RAM, it asserts one or two of the $\overline{\text{CAS}}$ outputs; this activates a group of dynamic

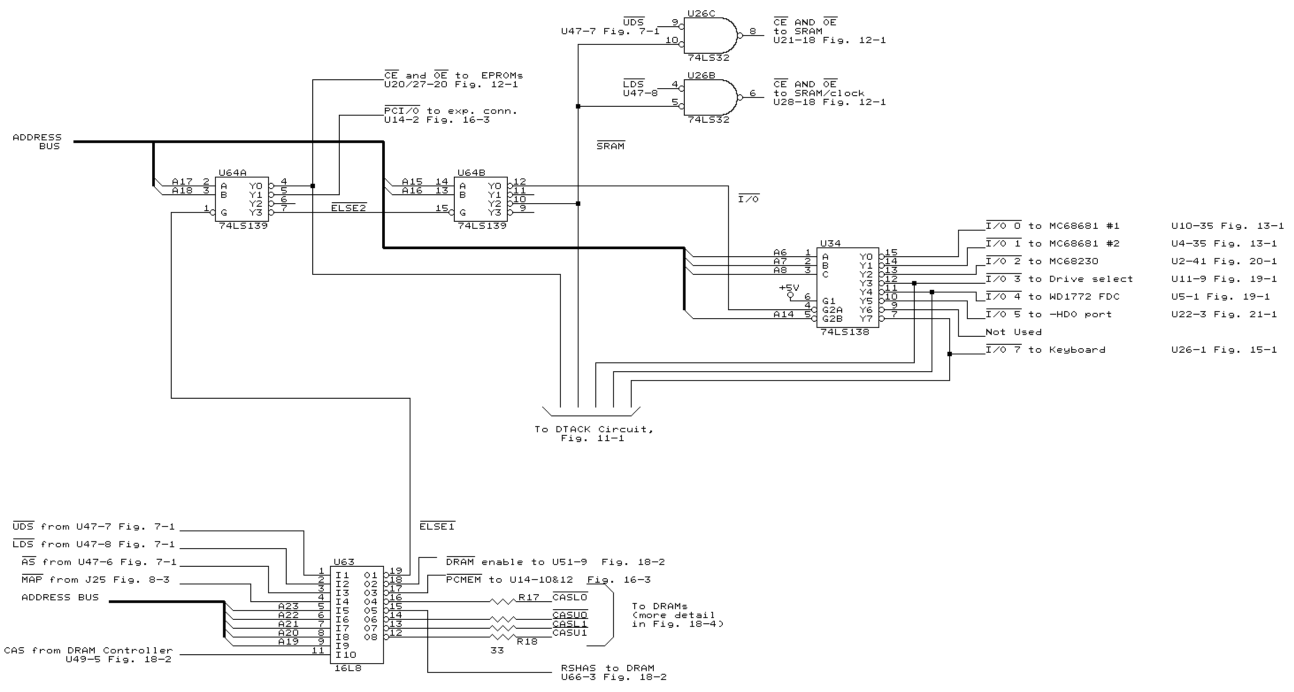

ADDRESS BUS
U64A
74LS139
U64B
74LS139
$\overline{CE}$ and $\overline{OE}$ to EPROMs U20/27-20 Fig. 12-1
$\overline{PCI/O}$ to exp. conn. U14-2 Fig. 16-3
$\overline{ELSE2}$
$\overline{UDS}$ U47-7 Fig. 7-1
U26C
74LS32
$\overline{CE}$ AND $\overline{OE}$ to SRAM U21-18 Fig. 12-1
$\overline{LDS}$ U47-8
U26B
74LS32
$\overline{CE}$ AND $\overline{OE}$ to SRAM/clock U28-18 Fig. 12-1
$\overline{SRAM}$
$\overline{I/O}$
U34
74LS138
+5V
$\overline{I/O\ 0}$ to MC68681 #1 U10-35 Fig. 13-1
$\overline{I/O\ 1}$ to MC68681 #2 U4-35 Fig. 13-1
$\overline{I/O\ 2}$ to MC68230 U2-41 Fig. 20-1
$\overline{I/O\ 3}$ to Drive select U11-9 Fig. 19-1
$\overline{I/O\ 4}$ to WD1772 FDC U5-1 Fig. 19-1
$\overline{I/O\ 5}$ to ~HD0 port U22-3 Fig. 21-1
Not Used
$\overline{I/O\ 7}$ to Keyboard U26-1 Fig. 15-1
To DTACK Circuit, Fig. 11-1
$\overline{ELSE1}$
$\overline{UDS}$ from U47-7 Fig. 7-1
$\overline{LDS}$ from U47-8 Fig. 7-1
$\overline{AS}$ from U47-6 Fig. 7-1
$\overline{MAP}$ from J25 Fig. 8-3
ADDRESS BUS
CAS from DRAM Controller U49-5 Fig. 18-2
U63
16L8
$\overline{DRAM}$ enable to U51-9 Fig. 18-2
$\overline{PCMEM}$ to U14-10&12 Fig. 16-3
R17
$\overline{CASL0}$
$\overline{CASU0}$
$\overline{CASL1}$
$\overline{CASU1}$
R18
33
To DRAMs (more detail in Fig. 18-4)
RSHAS to DRAM U66-3 Fig. 18-2

Fig. 10-1. The address decoder.

RAM ICs. $\overline{LDS}$ and $\overline{UDS}$ decide whether to use one of the $\overline{CASL}$ or $\overline{CASU}$ outputs (for either an odd byte or even byte or both), while A19 splits the megabyte into a lower 512K and an upper 512K. Incidentally the four resistors, parts of 33-ohm resistor packs R17 and R18, slightly slow down the rise and fall times of these signals. The $\overline{CASL}$ and $\overline{CASU}$ each go to eight dynamic RAM chips, and fast rise and fall times cause sharp signal edges which contribute to noise in the memory. The resistors reduce this noise and improve reliability.

3. If the address begins with the three bits 110, representing all addresses beginning with a hex C (1100) or hex D (1101), the $\overline{PCMEM}$ signal is asserted. This signal goes to the six PC-compatible expansion connectors. 1
4. If all five address lines are 1, representing all addresses beginning with the bits 11111 (addresses between \$F80000 and \$FFFFFF), the line labelled $\overline{ELSE1}$ is asserted; this signal implies that this address is something else besides dynamic RAM or memory on expansion slots.
5. Operation is a bit different if $\overline{MAP}$ is low; if so, then the dynamic RAM is disabled and the $\overline{ELSE1}$ signal is asserted instead of it. This substitutes ROM instead of RAM at address 0.

1 In keeping with the standards of Intel microprocessors, cards plugged into the expansion connectors can be addressed either as memory or as I/O. In the SK68K, both of these addresses appear as memory, but at different addresses.

In other words, the PAL splits the 16 megabyte address space of the computer into three main areas:

\$000000 - \$0FFFFF	$\overline{\text{DRAM}}$	dynamic RAM
\$C00000 - \$DFFFFF	$\overline{\text{PCMEM}}$	PC expansion memory space
\$F80000 - \$FFFFFF	$\overline{\text{ELSE1}}$	all else

Any other address simply doesn't get recognized by the address decoder.

When $\overline{\text{ELSE1}}$ is low, addresses in its range of \$F80000 - \$FFFFFF are further split up into three smaller groups by U64a, depending on address bits A18 and A17. U64 asserts one of its outputs when it is enabled (the $\overline{\text{G}}$ input is low); which output is asserted depends on the binary input on its B and A inputs. It asserts the Y0 output when it receives a 00, the Y1 output when it receives 01, and so on - it simply interprets the B and A inputs as a number between 0 and 3.

1. If A18 and A17 are 00, which occurs for addresses \$F80000 - \$F9FFFF, U64a asserts the $\overline{\text{CE}}$ and $\overline{\text{OE}}$ signals for both EPROMs, enabling them. Note how U64 has bubbles on the output to remind us that the outputs are active low.
2. If A18 and A17 are 01, which occurs for address \$FA0000 - \$FBFFFF, U64a asserts the $\overline{\text{PCI/O}}$ line. This signal goes to the six PC-compatible expansion connectors for accessing I/O addresses on plug-in cards. (These are the I/O addresses, not memory addresses.)
3. If A18 and A17 are 10, which occurs for address \$FC0000 - \$FDFFFF, U64a asserts its Y2 output, but this is not used.
4. If A18 and A17 are 11, which occurs for address \$FE0000 - \$FFFFFF, U64a asserts the $\overline{\text{ELSE2}}$ output.

As we're beginning to see, each IC in the chain in Fig. 10-1 uses the output of its neighbor to the left to narrow down the range of addresses it recognizes. Now U64b takes the $\overline{\text{ELSE2}}$ signal from U64a, and splits the \$FE0000 - \$FFFFFF range into two smaller groups, depending on address lines A16 and A15.

1. If A16 and A15 are 00, which occurs for addresses \$FE0000 - \$FE7FFF, U64b asserts the $\overline{\text{I/O}}$ signal which goes to U34.
2. If A16 and A15 are 10, which occurs for addresses \$FF0000 - \$FF7FFF, U64b asserts the $\overline{\text{SRAM}}$ signal which goes to the two static RAM (SRAM) ICs or the clock/calendar. Note how $\overline{\text{LDS}}$ and $\overline{\text{UDS}}$ also come into the picture to enable the upper 8 bits, lower 8 bits, or possibly both. This gating is done with U26b and U26c, and is a good example of the need for bubbles to simplify a circuit. In U26c, for example, if $\overline{\text{UDS}}$ is asserted and $\overline{\text{SRAM}}$ is asserted, then the $\overline{\text{CE}}$ and $\overline{\text{OE}}$ pins of U21 are asserted and this SRAM IC is enabled. Since both inputs and outputs of U26c have bubbles, all of these signals are low when asserted.
3. (If A16 and A15 are either 01 or 11, then U64b asserts one of its unused outputs.)

That brings us to U34, the last IC in the address decoder, which gets the I/O signal which is asserted for addresses $FE0000 - $FE7FFF. In addition, though, U34 also needs a low or zero on A14, so it actually only operates for addresses in the range from $FE0000 through $FE3FFF. U34 now looks at address lines A8, A7, and A6 to decide which of its outputs to assert. Like U64, U34 interprets the three-bit number on its C, B, and A inputs as a binary number between 0 and 7, and asserts Y0 through Y7 in response.

U34's job is to assign addresses to seven I/O devices: the two MC68681 DUART serial ICs, the MC68230 PIT parallel port, a drive select latch which controls the floppy drive, the WD1772 floppy disk controller IC, an optional WD1002 hard disk controller, or a PC-compatible keyboard.

For example, when A8, A7, and A6 are 000, U34 asserts the I/O0 line on Y0, which enables the first MC68681 DUART.

Now comes a difficult question - what address range does this DUART respond to? Following the circuit from left to right, we see that, for I/O0 to be low (selecting the first DUART),

(a) A23 through A19 must be 11111

(b) A18 and A17 must be 11

(c) A16 and A15 must be 00

(d) A14 must be 0

(e) A13 through A9 are unknown

(f) A8 through A6 must be 000

(g) A5 through A0 are unknown.

Thus we see that this address decoder does not really look at all the bits of the address bus - there are eleven bits unaccounted for (or ten bits when we realize that A0 doesn't exist.) Let's not worry about A5 through A0, since we will see that they do play a role later. But A13 through A9 are the problem.

Let's put the above bits in order, left to right, like this:

```
A23..........................A0
 |                            |
 1111 1110 00xx xxx0 00yy yyyy
```

The five unknown bits in the middle are labelled xxxxx, and the six unknown bits on the right are labelled yyyyyy. Clearly these x and y bits could be anything - all zeroes, all ones, or any combination of zeroes and ones.

Let's begin by assuming that the x bits are all zeroes, giving us

```
A23..........................A0
 |                            |
 1111 1110 0000 0000 00yy yyyy
```

Since the bits are already separated into groups of four, let's follow through and convert them to hexadecimal. The first four digits are clearly $FE00. If the y bits are all zeroes, then the last two digits are $00; if they are all ones, then the last two digits are $3F. This tells us that the DUART can be addressed in the range from $FE0000 through $FE003F. This is a total of

64 addresses, and is in fact the range of addresses that we use in programming it.

But the x digits need not be zeroes - they could just as well be 00001, which would make the complete address look like this:

```
A23..........................A0
 |                           |
 1111 1110 0000 0010 00yy yyyy
```

Again assuming that the y digits can be anything from all zeroes to all ones, this gives us an address range of $FE0200 through $FE023F. Continuing like this, we see that the DUART responds to many addresses:

$FE0000 through $FE003F
$FE0200 through $FE023F
$FE0400 through $FE043F
$FE0600 through $FE063F
$FE0800 through $FE083F

and so on, all the way up to

$FE3E00 through $FE3E3F

when the x bits are all ones.

This is all an example of *incomplete address* decoding, caused by the fact that the address decoder does not decode all the bits - some of the bits can be anything, and are usually called *don't cares*. Incomplete address decoding results in the use of more addresses than are actually needed.

The EPROM is another example of incomplete decoding. As we learned above, the EPROM is assigned addresses from $F80000 through $F9FFFF, a total of 128K bytes. But in a typical case, the EPROM may consist of a pair of 27128s, which provide a total of only 32K of EPROM, so that 96K of addresses (128K minus 32K) are wasted. What actually happens is that the 32K of actual EPROM appears in that 128K space four times. That is, the EPROM appears to take up the entire 128K, but on closer examination we see that there are four copies of the same data in that space. Such a loss of addresses in an 8-bit computer with a total of 64K of addresses would be unthinkable; in a computer having 16 megabytes of addresses, the loss of 96K is unimportant.

The same situation occurs with the I/O. Although each of the I/O devices may only require a few bytes, each takes up thirty-two 64-byte chunks of addresses, for a total of 2048 addresses.

10-2. Construction

Install the following components:

U63	16L8 PAL and its socket
R17 and R18	33-ohm resistor packs soldered directly to the board without a socket
U64	74LS139 dual decoder and its socket
U34	74LS138 decoder and its socket

C13, C67 and C70 0.1 µF disc capacitors

Finally, place the J25 jumper in position 1.

10-3. Testing

Recheck all connections and then apply power.

Although a really thorough test of the address decoder requires sophisticated equipment, we can nevertheless do a simple check with the aid of Fig. 10-2, which shows the waveforms output by the decoder circuit.

As you remember, the 68000 is still executing what it thinks is 4 million OR instructions, looping through memory once every four seconds. Think of Fig. 10-2 as showing what happens during those four seconds (there is a tiny bit at the right end of the figure which shows how everything repeats after those four seconds are up). During those four seconds, however, the 68000 is also counting up through 16 megabytes of memory. Thus the left of the figure corresponds to its trying to access location 0; the right end corresponds to its trying to read location $FFFFFF, which is at the top of the 16 megabytes; the midpoint corresponds to location $800000, the 8-megabyte point of memory. So Fig. 10-2 also represents the 16 megabyte range of memory, and some of the more important memory addresses are shown at the bottom.

Now let’s look at the $\overline{\mathrm{DRAM}}$ signal. Since dynamic RAM will comprise the first 1 megabyte out of the sixteen, the $\overline{\mathrm{DRAM}}$ signal goes low for the first 1/16th of the four seconds; that works out to one quarter of a second in terms of actual time. It is shown as a solid block in Fig. 10-2, however, because it does not stay low all that time. Like all other signals in the address decoder, $\overline{\mathrm{DRAM}}$ is asserted only when the $\overline{\mathrm{AS}}$ address strobe is on; since $\overline{\mathrm{AS}}$ keeps rapidly cycling high and low, so does $\overline{\mathrm{DRAM}}$. If you connect your LED probe to the $\overline{\mathrm{DRAM}}$ output, pin 18 of U63, you will note that the LED is on continuously (since the signal is high), but once every four seconds, the LED will get slightly dimmer for about a quarter of a second. An oscilloscope would show a continuous high, with a short burst of pulses once every four seconds.

Each of the other address decoder outputs can be checked in the same way except for the four $\overline{\mathrm{CAS}}$ outputs (which will show a constant high

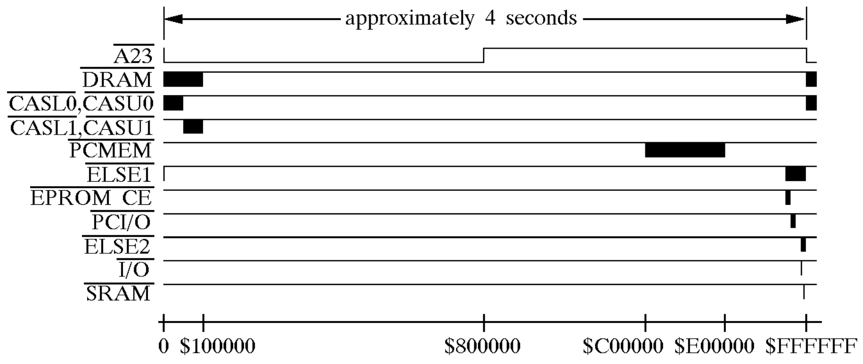


Fig. 10-2. Address decoder output waveforms.

because we are not supplying a CAS input to U63.) Some of the other outputs will be longer - such as $\overline{\text{PCMEM}}$ which will dim the LED for a 1/2 second - while most will be much shorter. The $\overline{\text{I/O}}$ and $\overline{\text{SRAM}}$ outputs will be very difficult to see since they only last about ten milliseconds, just barely long enough to flicker the LED if you watch carefully. The outputs of U34 are too short to see on our LED, although a pulse-catching logic probe or oscilloscope will show a slight flicker once every four seconds.

Chapter 11

The $\overline{\text{DTACK}}$ Circuit

Up until now, we have been generating an artificial $\overline{\text{DTACK}}$ with a temporary jumper; it's time to substitute a real $\overline{\text{DTACK}}$ circuit and see how it works.

11-1. Discussion

Ideally, each device, whether memory or I/O, should generate its own $\overline{\text{DTACK}}$ when it has finished an operation. In this way, the 68000 would know right away that it is time to continue to the next step. That is practical in some cases, such as dynamic RAM or some I/O devices, but in others it may be necessary to build a timer which just waits a certain time and then assumes that all is well. Thus the SK-68K system takes both approaches.

Fig. 11-1 shows the heart of the $\overline{\text{DTACK}}$ circuit. This circuit has eleven inputs, of which two (AS and CLK8) are used strictly for timing.

Five of the inputs come directly from the address decoder in Fig. 10-1. Three of these go to U37a: $\overline{\text{I/O4}}$goes low whenever the address decoder enables the WD1772 floppy disk controller; the EPROM $\overline{\text{CE}}$ signal goes low when the EPROM is selected, and $\overline{\text{SRAM}}$ goes low when the address decoder selects the static RAM. Whenever any of these three goes low, U37a provides a high output to U33. Note that U37a is a three-input "positive NAND" gate, but in this application it ORs three active-low signals and provides an active high signal whenever any of the inputs is asserted (low).

U33 is a quad D-type flip-flop, wired exactly the same as U76 in Fig. 9-1, so this time it is shown as a single block. It is clocked by CLK8, the 8 MHz clock signal, but most of the time the output of U37a is low, and so its four flip-flops are normally forced reset. That makes its $\overline{\text{3Q}}$ output (the $\overline{\text{Q}}$ output of the third flip-flop) normally high.

Now suppose the address decoder selects either the floppy disk controller, the EPROM, or the static RAM. The output of U37a then goes high,

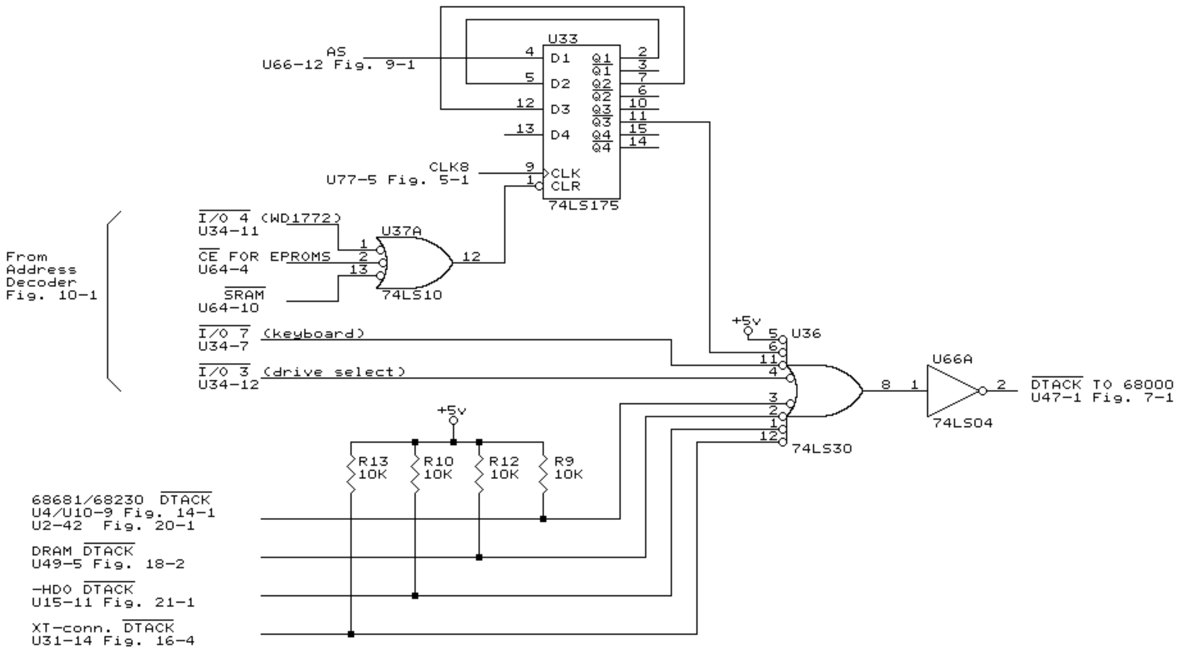


Fig. 11-1. DTACK generator circuit.

releasing the CLR signal on U33 and allowing it to start shifting. Since AS is high at this point, this high starts shifting through the flip-flops. After three CLK8 clock pulses, the 3Q output goes low. In other words, U33 acts as a delay, sending a low pulse to U36. Note that it is clocked by CLK8, not by MPUCLK. Even if you speed up the computer by using a faster MPUCLK, the delay in U33 will not change.

Now let's look at U36, an 8-input "positive NAND" 74LS30 which is used here as a "negative OR with a positive output" or, rather, to OR seven active low signals and produce an active high output (only seven inputs are needed, so pin 5 is permanently negated by being connected to +5 volts). One of these seven inputs comes from U33; one comes directly from the keyboard select line, I/O7, in the address decoder; and another comes from the drive-select enable line, I/O3, also in the address decoder. (The keyboard circuit and floppy disk drive select circuits are very fast and so I/O7 and I/O3, their select signals, immediately generate a DTACK without any other delay.)

The other four inputs to U36 come from other parts of the computer which we have not yet built or discussed. Each of these other parts generates its own DTACKwhen it is finished with a data transfer, and all of these are also ORed in U36. For now, however, we need resistors R9, R10, R12, and R13 to keep these four inputs high and prevent any undesired DTACKs from being generated.

U36 therefore acts as a large OR gate - whenever it receives a low pulse on any of the seven inputs, it sends a high to U66, which then finally asserts DTACK by sending a low to the 68000.

11-2. Construction

To construct this $\overline{\text{DTACK}}$ circuit, **REMOVE** the jumper between pins 1 and 14 of U66, which we have used until now to generate a fake $\overline{\text{DTACK}}$. Then install the following components:

U33	74LS175 and its socket
U36	74LS30 and its socket
R9, R10, R12, R13	10K 1/4-watt resistors

U37 and U66 have already been previously installed; the J25 jumper should still be in position 1 from our previous step; and at this point you should still have the Molex pins inserted in the U21 and U27 sockets, but no other extra jumpers.

11-3. Testing

Now turn on the power again and let's see what happens.

Nothing! Well, of course not. The problem is that the 68000 is still trying to execute 4 million OR instructions; it is getting those instructions from our Molex pins all right, but it is not getting $\overline{\text{DTACK}}$. Hence the $\overline{\text{BERR}}$ bus error circuit is timing out and halting the whole thing. You probably noticed that the HALT LED stays on and never even flickers.

Now move the J25 jumper from position 1 to position 2 and try again; you will see a slight flicker on the HALT LED about a second after you turn on the power (or force a reset by shorting J23), but it still lights.

Since the Molex shorting pins on U21 and U27 put all zeroes on the data bus, the start address which the 68000 gets just after a reset is also all zeroes. It therefore starts to execute a program which it thinks starts at address 000000. With J25 in position 1, low memory is supposed to be dynamic RAM; since there isn't any, there is no DRAM $\overline{\text{DTACK}}$ coming into U36; hence the 68000 quits almost immediately with a bus error. With J25 in position 2, however, the EPROM is supposed to be mapped into low memory. There isn't any EPROM, of course, but the address decoder and $\overline{\text{DTACK}}$ generator don't know that; hence they generate $\overline{\text{DTACK}}$as if the EPROM were there. The 68000, therefore, starts to execute its OR program until it passes the address where EPROM would normally end; then $\overline{\text{DTACK}}$ suddenly disappears and the system goes into HALT as a bus error appears.

Chapter 12

ROM and Static RAM

In order to do some useful work, the SK68K needs some RAM and ROM. The ROM contains a debugging program called HUMBUG and a simple Basic interpreter; the RAM is needed as a temporary memory to hold various data needed by HUMBUG.

12-1. Discussion

A fully configured SK68K system contains both static RAM (also sometimes called SRAM) and dynamic RAM (called DRAM). Static RAM circuitry is generally very simple and inexpensive, but only for small amounts of memory. Whenever a very large amount of memory is needed, dynamic RAM is the only economical alternative. We will examine dynamic RAM in a later chapter; for now, however, we will concentrate only on static RAM.

Since static RAM is so much simpler - needing just two ICs - installing a small amount of it at this time gives us a fast way of getting the SK68K operating. For that reason, HUMBUG and the ROM-based Basic are specially configured to use only the static RAM. We will add DRAM later, after the SK68K is at least partly operational; in that way, we will be able to use the remaining parts of the system to debug the DRAM circuitry if there are any problems. We will not actually need the DRAM until we are ready to run the disk system; running large programs of the type best suited for DRAM is really not practical until we have some means of saving and loading them on a disk.

Fig. 12-1 shows the circuitry for the EPROM and the static RAM. The 68000's data bus is 16 bits wide, but no one makes EPROMS and RAMS which have 16 data lines. The solution is to use two 8-bit-wide EPROMS and two 8-bit-wide static RAMs, each of which holds 8 of the 16 bits.

The two EPROMs are shown at the top, and the two static RAMs are shown at the bottom. Three kinds of EPROMs can be used, though both

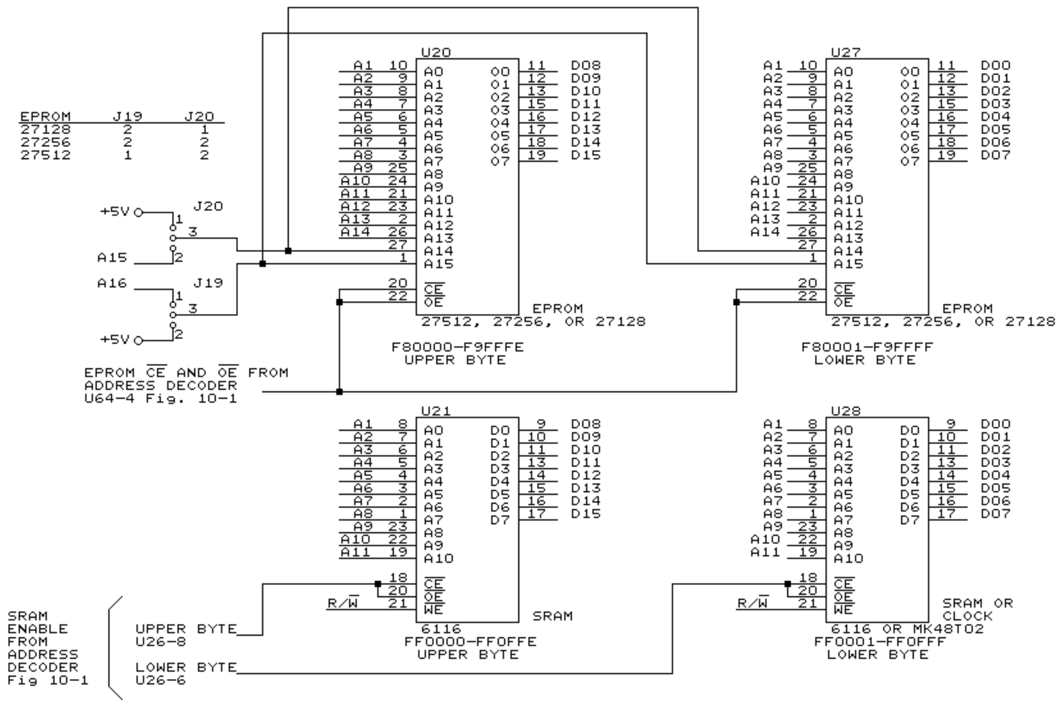


Fig. 12-1. EPROM and static RAM wiring.

EPROMs of a pair must be identical. A pair of 27128s holds a total of 32K bytes; a pair of 27256s holds 64K bytes; a pair of 27512s holds 128K bytes. Any one of these would be big enough to hold HUMBUG and Basic, but the prices on memory ICs have been so unstable that your SK68K may contain any one of these - it depends on which is cheapest at the moment. Jumpers J19 and J20 simply determine which kind of EPROM can be used, because larger EPROMs require more address lines.

The two static RAMs are shown at the bottom. The simplest version of the SK68K uses a pair of 6116s, which provide a total of 4K bytes of memory. But it is possible to replace one - or both - with a Mostek MK48T02, which also contains a static RAM and has the same pinout as a 6116, but has two additional features - it contains a built-in clock/calendar (which replaces the top 8 locations of the RAM), and it has a built-in lithium battery which powers both the clock and the RAM when the computer is turned off. The battery is rated for at least 31,000 hours of operation, or slightly more than 3-1/2 years. In actual use, it would probably last longer since the battery only powers the clock when the computer is off. (Note that the MK48T02, if used, must be inserted into U28, since the clock software in SK*DOS expects to find it in the lower byte.)

As to the wiring, the two left ICs (left in Fig. 12-1; physically, they are really located toward the back of the actual board) connect to D8 through D15; the two right ICs connect to D0 through D7. Thus the two left ICs handle what is called the *upper byte*, whereas the two right ICs hold the *lower byte*. This is sometimes confusing and needs some explanation. When the 68000 stores a word (two bytes) from an internal register into memory, the left byte (the more significant byte, also called the upper byte) goes into memory first, and is always in an even-numbered location. The right byte (least significant or lower byte) goes into memory next, and goes into the next higher odd location. For example, the number $1234 might be stored as $12 in location $3500 and $34 in location $3501. This is the more logical way to do things, but it is confusing for two reasons: (1) the upper byte is actually stored in the lower address, and (2) this is the opposite of what Intel processors do.

Both EPROMs are controlled by the same $\overline{\text{CE}}$ (chip enable) and $\overline{\text{OE}}$ (output enable) signals, whereas there are separate enable signals for the static RAMs. Reading both EPROMs at the same time does no harm - if the 68000 only wants one byte, it simply ignores the other half of the data bus. But writing to the static RAM requires two control lines to make sure that writing to one RAM does not store garbage in the other. Note also the difference between $\overline{\text{CE}}$ and $\overline{\text{OE}}$ - when a read is done from RAM or EPROM, $\overline{\text{CE}}$ enables the IC and starts the read process, but no output gets to the bus until $\overline{\text{OE}}$ is asserted. In many systems, $\overline{\text{CE}}$ is used to put the entire chip into a low power mode when not being used; in the SK68K we simply control both together so it always switches into low power mode after every access.

Note that on all four memory ICs, the IC's A0 connects to A1 on the address bus, A1 connects to A2, and so on. Part of the reason is simply that A0 does not exist on the address bus; part comes from the way memory is addressed on the SK68K. Consider the static RAM, for example, which is addressed starting at address $FF0000. The bytes of this RAM are stored as follows:

$FF0000 is in U21 location 0
$FF0001 is in U28 location 0
$FF0002 is in U21 location 1
$FF0003 is in U28 location 1
$FF0004 is in U21 location 2
$FF0005 is in U28 location 2

and so on, with all even addresses (the 'upper byte') in U21, and the odd addresses ('lower byte') in U28. The address lines are 'shifted over one bit' because shifting a binary number to the right by one bit divides it by two. For example, location $FF0008 ends with the bits 1000; it would be in location 100 of U21. Location $FF0009, on the other hand, ends with the bits 1001; it would also be in location 100, but in U28. Thus the last bit of an address (0 in $FF0008, 1 in $FF0009) tells us which IC it is stored in, while the preceding bits specify the location in that IC.

12-2. Construction

Now that we understand how the EPROM and static RAM circuitry works, let us actually connect it. Remove the Molex pins from the U21 and U27 sockets, and install the following components:

U26	74LS32 and its socket
C12 and C66	0.1 µF disc capacitors

U20	EPROM marked ’upper’ and its socket
U27	EPROM marked ’lower’
U21	6116 2Kx8 static RAM
U28	6116 2Kx8 static RAM and its socket
J19 and J20	three-pin header strips

Examine the two EPROMS and position the J19 and J20 jumpers to match the type of EPROM used, following this table:

EPROM	J19	J20
27128	2	1
27256	2	2
27512	1	2

Place jumper J25 in position 2 to address the EPROM starting at location 0.

12-3. Testing

Now turn on the power.

The important sign that all is well is that the HALT LED goes off after about a second. If not, then go back a step, recheck that all worked before, and then recheck all connections and parts installed since then.

The design of 68000 computers is such that they tend to halt if something goes wrong. If the HALT LED is off, that is a pretty good sign that nothing major is wrong, even though the computer is still not fully operational.

Use the LED probe to check a few signals. First of all, all data bus lines should have pulses (indicated by the LED dimming when you connect to the line). Likewise, all address lines from A1 through A18 should show lots of pulses, whereas A19 through A23 should not dim the LED at all. If the HUMBUG program in the EPROM is running, it is accessing mostly itself (EPROM locations $F80000 and above) and static RAM (locations $FF0000 and above). Since both $F8 and $FF start with the five bits 11111, we would expect the five highest bits of the address bus - A23 through A19 - to be constantly high, even though the others may change.

You can confirm this by testing the outputs of the address decoder. You will note a lot of pulses, indicated by the LED dimming, on U63 pin 19, which decodes all addresses above $F80000. U64 will have pulses on pin 4 (which selects the EPROM), pin 7 (which drives U64b), and pin 10 (which selects the static RAM.)

What you don’t see on the LED (though a good scope will reveal them) are tiny low pulses on A23 through A19, as the 68000 is trying to access some error vectors because it senses a bus error - no DUART is installed. Likewise, you can see some tiny pulses on U64 pin 5, as HUMBUG is desperately trying to access the expansion slots in the hope that there is a video board there to report the error on.

Chapter 13

The Magic Moment: First Signs Of Life

At this point, we can finally get the computer to do something useful - we call it the Magic Moment when it shows the very first signs of Life.

13-1. Discussion

When you turn on the power (or short the reset pins at J23), the HUMBUG monitor program in the computer's ROM tries to initialize the input and output ports, makes a list of what options you have installed, and emits a "beep-boop" sound from the speaker.

The MC68681 DUART at U10, along with the 3.6864 MHz oscillator at U3, is used mainly for the serial ports. But the DUART also has an internal counter and timer, which is used to generate the tones for the speaker. This

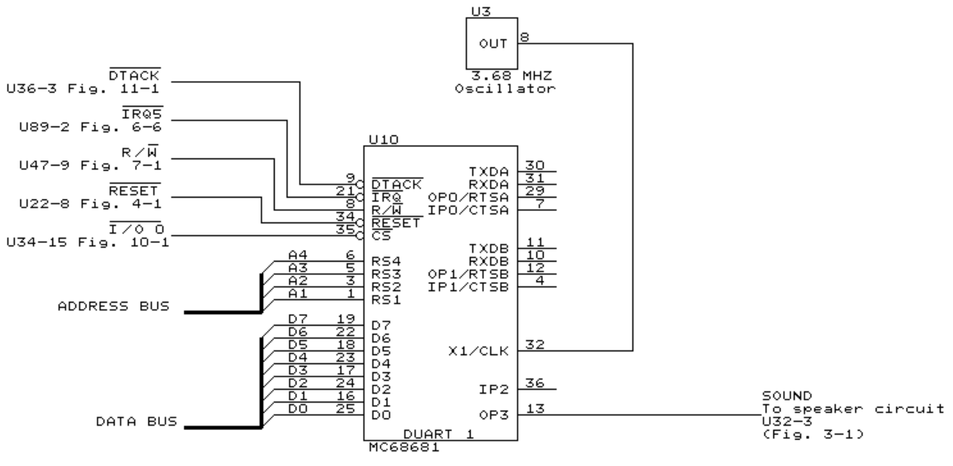


Fig13-1. DUART1 sound generator.

circuitry is shown in Fig. 13-1, though only one small wire - the connection to pin 13 of U10 - is dedicated to the speaker circuit.

The main IC in this circuit is U10, the DUART. This is a *Dual Asynchronous Receiver-Transmitter* which handles the conversions between the parallel data as it is sent along the data bus, and the serial data as it travels along the serial link to the terminal or printer. Though it carries the "UART" letters in its name, it is much more than the traditional UART, which only handles the conversion to and from serial data. Not only does the DUART contain two internal receiver/transmitter ports, but it also contains a multi-function 6-bit parallel input port, an 8-bit output port, and a 16-bit programmable timer/counter. We will discuss the other capabilities of the DUART in upcoming chapters. The DUART does use bit 3 of the output port (pin 13, called OP3) and the programmable counter/timer to drive the speaker circuitry, as shown in Fig. 13-1.

The connections to the rest of the computer (on the left of U10) are fairly straightforward. The clock signal comes from U3, a 3.6864 MHz oscillator module. The DTACK output connects directly to the DTACK gate, U36; the IRQ output connects to IRQ5, the level 5 IRQ input; R/W and RESET connect to the same points in the rest of the system, and CS, the chip select input, connects to the I/O0 point (U34-15). As we have already discovered, this address decoder output goes low for addresses in the range from $FE0000 through $FE003F.

Finally, data bus pins D0 through D7 connect to the lower eight bits of the data bus, while the four register select lines, RS1 through RS4, connect to A1 through A4, respectively. Since there are four register select lines - four address lines - the DUART occupies 16 addresses in the 68000's memory space. But since only the lower eight bits of the data bus are involved, these sixteen addresses are all odd, so the DUART is addressed at $FE0001, $FE0003, $FE0005, and so on, up through $FE001F.

Internally, a DUART contains almost two dozen registers, some of which can be read by the 68000, and some of which can be written into by the 68000. It takes Motorola about 70 pages to explain the DUART in their MC68681 data 'sheet' (Motorola publication ADI-988, available from Motorola distributors or from Motorola Semiconductor Products, 3501 Ed Bluestein Blvd., Austin TX 78721), so there is no way to do it justice here. We strongly recommend that you obtain that manual.

13-2. Construction

Install the following parts:

U3	3.6864 MHz oscillator module (possibly in a special socket)
U10	MC68681 DUART and its socket

Then connect a speaker to pins 1 and 4 of J18. The small speaker which comes with XT or AT clone cabinets is best; any other speaker or earphone will do, but choose a cheap one. The circuit puts a constant DC current through it which, though it won't harm the speaker, will certainly not do much good to an expensive hi-fi speaker system.

13-3. Testing

Now turn on the power; about a second later, the HALT LED should go off, and another second later you should hear a beep-boop from the speaker. This is a signal from the HUMBUG program that it is up and running, and is the first big indication that the computer is really capable of running a real program - even the slightest problem would prevent HUMBUG from sounding this dual tone.

Now that a real program is running with no errors, the signals in the address decoder will change slightly from before. HUMBUG is currently running a loop which continuously checks for the presence of a keyboard; this loop runs entirely in EPROM and calls the first DUART, so you will see pulses on the EPROM chip select line, U64a pin 4; the I/O line on U64b pin 10; and the I/O0 line, U34 pin 15. You will no longer see pulses on the SRAM chip select line.

13-4. In Case Of Difficulty...

Even if your system is working properly and emits the beep-boop sound from the speaker, read the following material so as to get a good idea of what the system is doing and how one would proceed to troubleshoot it if necessary.

In case of difficulty, we start with all of the obvious possibilities.

First, using a good magnifying glass, carefully examine all solder joints to make sure all connections are soldered, but no shorts exist between adjacent lands on the board. Try to check that all IC socket pins really go through the board to the bottom side - sometimes a pin bends under the socket and never makes it through the board. Make sure that all IC's point in the same direction - pin 1 toward the left rear corner - and that each socket contains the correct IC. Check also that all capacitors and resistors are in their correct places. It might also be a good idea to check power supply voltages.

While on the subject of power supply voltages, note that the wide copper strip going all around the edge along the bottom side of the board is ground, but the edge conductor on the top of the board is +5 volts. Hence if you mount the board with metal hardware, you will short +5 volts to ground. Moreover, it is possible to short out the power supply by clipping a scope or meter ground lead to the edge of the board. Be careful in this regard.

Next, check that all the correct components are installed, but no others. Extra oscillators, resistors and capacitors can stay, but any extra ICs should be unplugged during troubleshooting, as they can cause undesirable results.

Check also all jumpers and other connections. The speaker should be connected to the two outer pins of J18; J25 should connect the center pin to pin 2; J24 should connect the center pin to pin 1; J19 and J20 should be set according to the table in Fig. 12-1.

If all these simple tests reveal no problems, then it's time for more drastic action. At this point, there are two ways to go - the brute-force method is to go back to Chapter and then try to retrace our steps. But there is another, more elegant method which lets us use the 68000 to help us debug. Let's describe the latter.

Let's review how the 68000 works. When it wants to access memory or I/O, the 68000 outputs an address on the address bus, along with the appropriate control signals on $\overline{\text{UDS}}$, $\overline{\text{LDS}}$, etc. When the memory or I/O is finished with the requested operation, it sends back a $\overline{\text{DTACK}}$ or data transfer acknowledge signal, which tells the 68000 that the operation is finished and it can continue to the next step. If, for some reason, an operation cannot be completed within a specified time, then the bus error circuit sends a $\overline{\text{BERR}}$ signal to the 68000, which attempts to recover. If the recovery procedure results in another bus error, then the 68000 simply halts and the HALT LED goes on. This is most likely what is happening if your HALT LED either stays on continuously, or flickers about 1 second after you turn on power or reset the system.

To proceed, disable the $\overline{\text{BERR}}$ circuit by pulling out U76 and connecting a 10K resistor from U76 pin 14 to pin 16. This forces $\overline{\text{BERR}}$ always high, so that a bus error will never be received. (We have soldered two Molex pins to the leads of the 10K resistor, so we can just insert it into the socket instead of U76. Don't force the leads of the resistor into the socket, as they will stretch the socket pins.)

Next, disable the $\overline{\text{DTACK}}$ circuit by pulling out U36 and connecting a 330-ohm resistor from U36 pin 8 to pin 7. Since U66 is installed (we can't remove it because it is used in the reset circuit), this forces the $\overline{\text{DTACK}}$ input into the 68000 high so it never receives a $\overline{\text{DTACK}}$.

Now turn on the power. The very first thing the 68000 tries to do upon power up is to get a stack address and a starting address from locations 000000 through 000007, which are at the very beginning of the ROM and which contain the following data:

Address	Data	
00000000	00FF0FEA	Address for the stack
00000004	00F800C0	Starting address of HUMBUG

A bit of explanation is in order. Although addresses inside the 68000 consist of 32 bits (and thus are expressed as eight hex digits in the above), the actual 68000 address bus only consists of 24 lines (counting A0 even though it only exists internally within the 68000). Thus the first two hex digits of addresses are generally 00.

Furthermore, the 32-bit address 00FF0FEA is too long to fit into a memory location which can only hold an 8-bit byte. It is therefore stored in four consecutive locations, namely locations 00000000 through 00000003. That explains why the address of the second line in the above is 00000004 - it is the next available location after location 00000003.

Finally, although we have a 4-byte address stored in four 1-byte locations, the 68000 can read or write two bytes at a time over its 16-bit data bus. It must therefore read the 4-byte address in two parts.

So what happens is this: when the 68000 tries to read the stack address from memory, it outputs address 000000 on the address bus, makes both $\overline{\text{UDS}}$ and $\overline{\text{LDS}}$ low to signal that it is trying to access both an upper byte and a lower byte, makes the $\overline{\text{AS}}$ address strobe low to activate the address decoder, and makes R/$\overline{\text{W}}$ high to indicate it is reading. The address decoder decodes address 000000, and sends a low to both EPROMs to enable them. The EPROMs, in turn, output the contents of the first two bytes (the number $00FF, the first half of the $00FF0FEA stack address) to the data bus, which sends this data to the 68000. The 68000 receives the data, and now waits for $\overline{\text{DTACK}}$ so it can continue.

But we've disabled $\overline{\text{DTACK}}$! Thus the microprocessor just sits there, waiting. Take a scope, logic probe, or even a meter, and look at its pins. You will see the address 000000 on the address bus, 00FF on the data bus, both $\overline{\text{UDS}}$ and $\overline{\text{LDS}}$ low, $\overline{\text{AS}}$ low, and R/$\overline{\text{W}}$ high. (It may be useful to place a label on the top of the 68000 and label each pin). You will also see lows on the $\overline{\text{CE}}$ and $\overline{\text{OE}}$ pins of both EPROMS, and a low on IC64a pin 4. In other words, even with fairly simple test equipment, we can check out the circuit to make sure all the right signals are there. Table 14-1 is a complete list of what you should find on every pin of the 68000 at this point - check to see that this is so, and trace the signals if not. (The table refers to lines as being either High or Low. A high should be +3 volts or more; a low should be below 0.4 volt; if you see any lines which are between those levels, look for a possible short circuit which is connecting a high to a low and producing an in-between voltage.)

(Here's an example of how to interpret strange results. Suppose all the signals on the 68000 are correct except for the sixteen data bus lines. If the data bus reads $FF00 instead of $00FF, are the EPROMs interchanged? If the data bus is completely different, are the EPROMs selected (lows on their $\overline{\text{OE}}$ and $\overline{\text{CE}}$ inputs)? If yes, are they getting the correct address? Is U64 pin 4 outputting a low? If not, is U64 getting lows on pins 1, 2, and 3? As long as the processor is totally stopped, tracing signals through is easy.)

If everything seems to be normal, the next step is to pulse $\overline{\text{DTACK}}$ low so the 68000 will proceed to its next step. This pulse must be wide enough so the 68000 will recognize it, and yet narrow enough so that the 68000 will only go forward one step and no more. The timing is thus touchy, but we have found that taking a discharged 0.001 µF capacitor (short its leads together first to discharge it) and momentarily connecting it from U36 pin 8 to pin 14 works quite well. Pin 8 is being held low by the 330-ohm resistor added a few paragraphs ago, but the 0.001 µF capacitor pulls it high for about a quarter of a microsecond; this is inverted by U66a into a low $\overline{\text{DTACK}}$ pulse.

As a result, the 68000 now tries to read the next two bytes of the stack address from memory, so the address bus should contain the address 000002, and the data bus should have the number 0FEA - the second half of the address 00FF0FEA. There is always the possibility that the pulse from the capacitor may be too wide or too narrow, or that more than one pulse may be generated, in which case you may either stay at the same step, or may skip ahead several steps. In that case, refer to Table 14-2, which is a list of the addresses and data that should exist on the address and data bus during the first dozen or so steps.

Looking at the address and data busses in this way should make it possible to spot address or data bus shorts or opens.

Assuming these results are normal, the next step is to remove the 330-ohm resistor, replace U36, and restart the system. The computer will now start executing HUMBUG from the beginning, going at high speed through the steps we were tracing one-by-one earlier. But any time that it tries to access a memory or I/O address that either doesn't exist or that is not properly working, it will fail to get DTACK; since we are still forcing BERR high with a 10K resistor, the 68000 will therefore stop and we can again look at the address bus to see where.

Since HUMBUG tries to compile a list of installed hardware, it intentionally tries to access addresses that may not yet exist on your system. In order, these are $FE0089 (to see whether a 68230 exists), $FE000B (to check for DUART 1), $FE004B (to check for DUART2), and either $C00001 or $D60001 (to check for the existence of PC/XT-compatible connectors). Thus the first address you see on the bus should be $FE0089 (though, of course, we cannot see A0. Instead, LDS will be low or on, while UDS will be high or off, to signify that the address is $FE0089 and not $FE0088.) If the 68000 stops with any other address on the address bus, the 68000 is trying to access some location that does not work or does not exist.

Once you see the $FE0089 address of the 68230, you may pulse $\overline{\text{BERR}}$ low once to get the 68000 to continue until the next bus error. This can be done with the same 0.001 µF capacitor as before, except this time connect it between U76 pin 14 and U76 pin 8, which is ground. As before, you may skip ahead a few extra steps if the pulses introduced by the capacitor are not quite right, in which case just start over.

Note that your computer should not stop at address $FE000B, since you have already installed DUART1 at U10. If it does, then there is something wrong with the DUART1 circuit.

TABLE 13-1. Signals on the 68000 pins as it waits for its first $\overline{\text{DTACK}}$

PIN NO.	STATE	SIGNAL NAME	DESCRIPTION	COMMENTS
52-50 and 48-29	Low	A23-A1	Address bus	$000000
54-61	Low	D15-D8	Data bus	00
62-64 and 1-5	High	D7-D0	Data bus	$FF
6	Low	$\overline{\text{AS}}$	Address strobe	on
7	Low	$\overline{\text{UDS}}$	Upper data strobe	on
8	Low	$\overline{\text{LDS}}$	Lower data strobe	on
9	High	$\overline{\text{R/W}}$	Read/$\overline{\text{write}}$	Read
10	High	$\overline{\text{DTACK}}$	Data Xfer acknowledge	Forced to off

TABLE 13-1. Signals on the 68000 pins as it waits for its first $\overline{\text{DTACK}}$

PIN NO.	STATE	SIGNAL NAME	DESCRIPTION	COMMENTS
11	High	$\overline{\text{BG}}$	Bus granted	off
12	High	$\overline{\text{BGACK}}$	BG acknowledge	off
13	High	$\overline{\text{BR}}$	Bus request	off
14	High	Vcc	+5 volts	power
15	Pulses	CLK	8 MHz clock	clock pulses
16	Low	GND	Ground	
17	High	$\overline{\text{HALT}}$	Halt	Not halted
18	High	$\overline{\text{RESET}}$	Reset	Not reset
19	High	$\overline{\text{VMA}}$	Valid Memory Address	off
20	Pulses	E	E clock	clock pulses
21	High	$\overline{\text{VPA}}$	Valid Peripheral addr	off
22	High	$\overline{\text{BERR}}$	Bus Error	Forced to off
23	High	$\overline{\text{IPL2}}$	Interrupt inputs	
24	High	$\overline{\text{IPL1}}$	"	no interrupts
25	High	$\overline{\text{IPL0}}$	"	
26	High	FC2	68000 Function Code	Currently in
27	High	FC1	"	supervisor
28	Low	FC0	"	program mode
49	High	Vcc	+5 volts	power
53	Low	GND	Ground	

TABLE 13-2. Address And Data Bus Contents During The First Few Memory Accesses

ADDRESS	DATA	EXPLANATION
000000	00FF	Initial stack address = $00FF0FEA
000002	0FEA	
000004	00F8	HUMBUG starting address = $00F800C0
000006	00C0	
F800C0	4EF9	JUMP (op code 4EF9) to $00F80126 instruction
F800C2	00F8	
F800C4	0126	
F80126	4239	A CLR.B instruction
F80128	00FF	
F8012A	002D	

TABLE 13-2. Address And Data Bus Contents During The First Few Memory Accesses

ADDRESS	DATA	EXPLANATION
F8012C	4239	Another CLR.B instruction
F8012E	00FF	
F80130	0C85	
F80132	48F9	A MOVEM instruction
F80134	7FFF	
F80136	00FF	
F80138	0C0E	

Chapter 14

Serial Interface

We are now at the stage where we are ready to install the I/O (input/output) circuitry to allow us to communicate with the computer via a keyboard and some sort of video display.

14-1. Discussion

There are two ways of communicating with the SK68K:

1. If you already have either a computer terminal (such as a Televideo or Soroq or whatever), or else a computer which can emulate a terminal (using a communications program), then it's easy - we install a serial port on the SK68K by adding two ICs and a connector, and communicate with the SK68K from the terminal via a serial port. This is discussed in this chapter.

2. The other method is to take advantage of the fact that we can use PC-compatible clone components. By installing the keyboard interface and PC-type bus connectors, we can then plug in a clone keyboard and either a monochrome video board or a CGA color board (with, of course, the appropriate monitor). This is discussed in Chapters 15 and 16.

Even if you do not use the serial interface at this time, we will still install it since we have already installed the DUART anyway, and since it is potentially useful to drive a serial printer.

Fig. 14-1 shows the circuitry for the four serial ports that can be installed on the board, although the bottom two ports are optional and seldom used. Since R9, the 3.6864 MHz oscillator at U3, and the MC68681 DUART at U10 have already been installed, at this point we need only add U29, U30, J21, and J22 to complete the top two ports.

As described in the last chapter, the DUART (U10) has a number of functions in the system. In addition to generating the tones for the speaker,

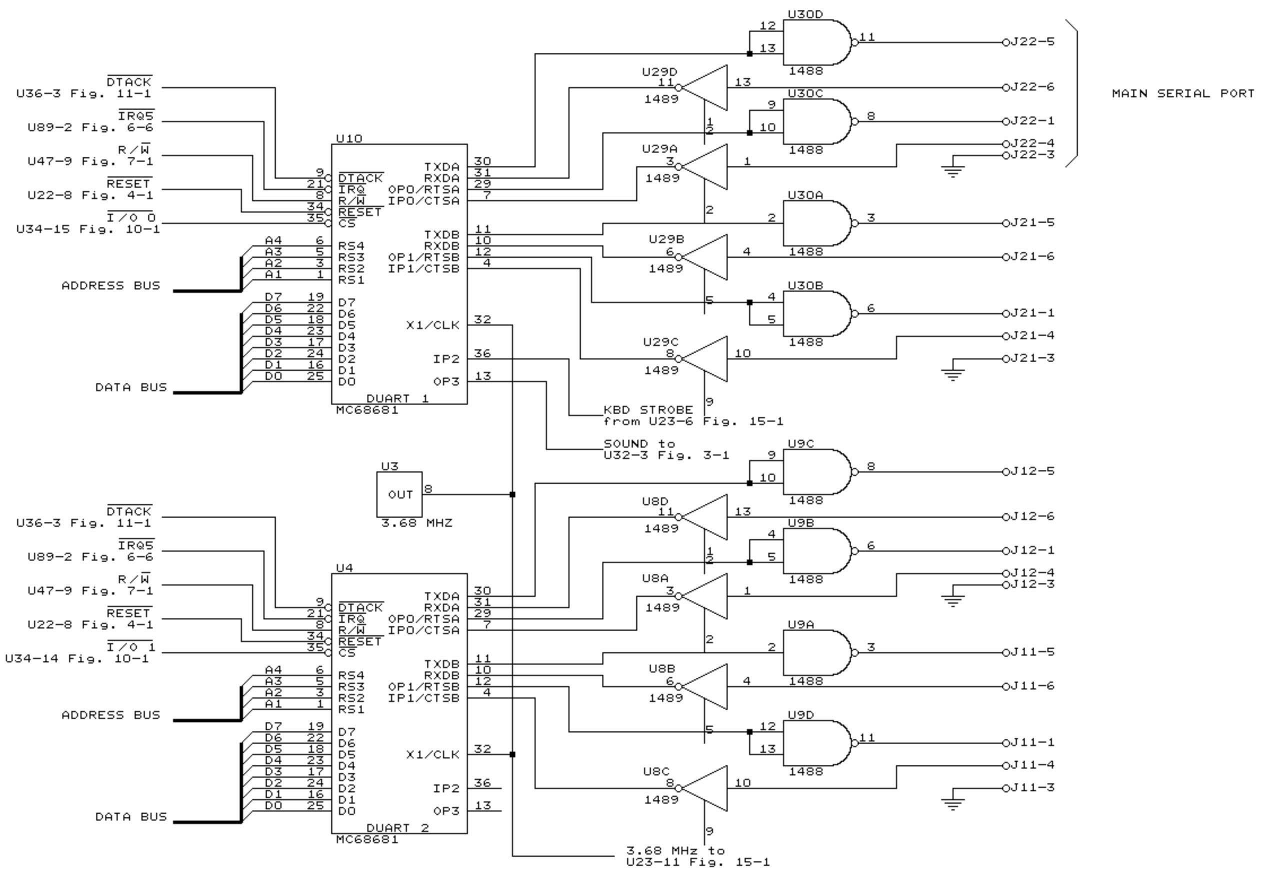


Fig. 14-1. Serial port circuitry.

it interfaces the two serial ports, and even handles the interrupts for many of the system components.

Since the DUART connects to the data and address busses, its internal registers appear in the 68000’s memory space as memory locations. It contains two serial ports, called port A and port B. The following addresses are for port A, the primary port used for a terminal:

$FE0007 is the data register. Sending a byte to this address outputs it to the first serial port, via J22; a character input from the port can be read at the same address.

$FE0003 can be read to determine whether the DUART is ready to send or receive a character. The bits at this address are numbered from 0 to 7, with bit 7 on the left and bit 0 on the right. In most cases, bits 0 and 2 are the most important. If bit 0 is a 1, then the DUART has received a character from the serial port, and the character can be read from $FE0007. If bit 2 is a 1, then it is ready to output a character to the serial port, and you should store that character into $FE0007.

Writing to $FE0003 selects the baud rate for the port. The HUMBUG software recognizes the baud rate of your keyboard and automatically sets the DUART baud rate to match (it supports only 300, 600, 1200, 2400, 9600, or 19200 baud), but the baud rate can be changed by placing a different number into address $FE0003. The allowable values are

Baud rate	$FE0003 value
110	$11
300	$44
600	$55
1200	$66
2400	$88
4800	$99
9600	$BB
19200	$CC

Addresses $FE0005 and $FE0015 can cause problems. The Motorola DUART data sheet lists these as "Do Not Access - This address location is used for factory testing of the DUART and should not be read. Reading this location will result in undesired effects and possible incorrect transmission or reception of characters. Register contents may also be changed." Accidentally reading this location may cause your SK68K system to crash.

14-2. Construction

Now mount the following parts on the board:

U30	1488 TTL-to-RS232C converter and its socket
U29	1489 RS232C-to-TTL converter and its socket
J21 and J22	two six-pin header strips

Cut off one center pin and position each header so the cutoff pin is on the side closest to U30; see Fig. 14-2 for details.

The two ICs do the voltage conversion between the DUART, whose inputs and outputs are TTL-compatible (0 is approximately 0 volts, whereas a 1 is about 5 volts), and the serial port wiring (which uses RS-232C levels of about +12 volts for a 0 and about -12 volts for a 1).

As Fig. 14-1 shows, there are four signal leads and one ground lead for each port. On the main port - port A, the one connected to J22 - TXDA is the *transmitter data line*, which sends serial data from the SK68K to a terminal, and RXDA is the *receiver data line*, which receives data from the terminal. RTSA is a *request to send* line which can be used to tell the terminal (or other device connected to the serial port) that the computer is asking for data. Finally, CTSA is a *clear to send* line which can be used by the terminal or other device to tell the computer that it is OK to send data to it. Unless you run special software, regular SK68K software ignores the RTS and CTS lines.

Fig. 14-2 shows how to connect a terminal to the six-pin header. On the terminal end, you will need a DB-25S connector with wiring to pins 2, 3,

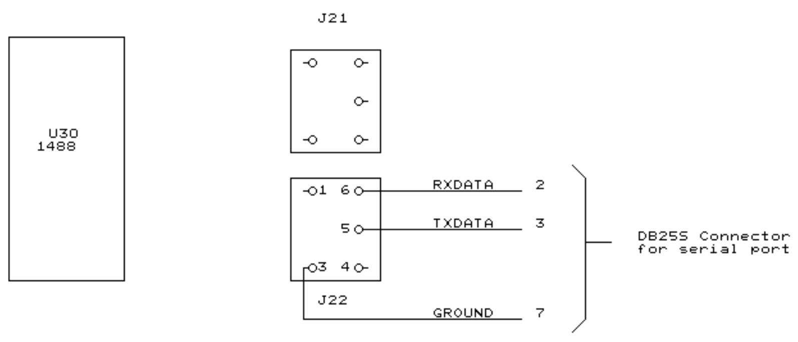


Fig. 14-2. Serial connector wiring.

and 7 as shown. On the computer end, you will need a special connector made by Berg and others (and supplied as part of the SK68K kit). First, crimp (or lightly solder) a Berg 47747 pin on the end of each wire, then insert the three pins into the top end of a Berg plastic shell number 65043-034. Finally, insert a 65307-001 key (a plastic insulating pin) into the hole corresponding to the cutoff pin (labelled with an X in Fig. 4-2) so as to make sure that the connector fits over the header just one way. Don't insert the extra pins into the remaining holes, as they cannot be connected to or removed once installed in the shell.

14-3 Testing

This section assumes that you have a serial terminal, or at least a computer which can emulate a terminal by using a communications program. If you do not have such a device, then skip ahead to Chapter 15.

Connect the DB-25S to the terminal, the Berg connector to J22 (not J21!), and turn on the power. Wait until the beep-boop tone is finished, and then press the RETURN (or ENTER) key on the terminal once or twice. If all goes well, the terminal should display the HUMBUG program signon message and then the prompt.

If the signon message does not appear, it's likely that there is a minor problem with the terminal wiring. First, set the terminal to either 300 or 1200 baud. Then, with a scope or meter (don't use the LED probe from J14-1 or you may damage U32) check the voltage on the RXDATA and TXDATA lines at J22; they should both be negative, between -3 and -15 volts. If the TXDATA line is negative but RXDATA is not, then try swapping the two connections at pins 2 and 3 of the DB-25S connector - both the computer and terminal may be sending data to the same line. If both are negative, check that pressing a key on the terminal makes the voltage on the RXDATA line swing from negative to positive and back; a scope will show this quite clearly, whereas a meter may just show a slight amount of wavering. A lack of this signal means that the terminal is not sending the RETURN code to the computer. Then check the TXDATA line; if it has a signal but the terminal displays nothing, then the terminal is not properly receiving its data. Many terminals require that several pins of their RS-232C connector be properly strapped together (usually pins 5, 6, 8, and 20) before they work.

Apart from possible problems with U29 and U30 (which may not be passing either the received or transmitted signal), there are very few other things which can go wrong at this point since all of the remaining parts of this circuit (including U3 and U10) are already required to make the speaker go beep-boop. (But check U3 anyway - if it is defective, U10 might still be receiving just enough noise that the speaker circuit works yet the serial port does not.)

Chapter 15

The PC-compatible Keyboard

PC-compatible keyboards are very different from the average computer keyboard in that they contain quite a bit of internal intelligence (including a buffer which stores data which the computer hasn't yet received), yet they do not generate an actual ASCII code for each key press. Such keyboards require a fairly complex interface, shown in Fig. 15-1, as well as a fairly complex program to process their output.

15-1. Discussion

The keyboard connects to J9 via a 5-pin DIN connector which sends ground, +5 volts, and the $\overline{\text{RESET}}$ signal to the keyboard, and receives the CLOCK and DATA signals from the keyboard. Note, however, that both CLOCK and DATA are bidirectional since the SK68K can send signals back to the keyboard via the same lines.

When the computer is turned on or reset, the BRESET input to U32e-11 (which is the same as RESET but buffered by U17) is inverted into a low and sent back to the keyboard on the CLOCK line; this causes the keyboard to reset. (Those keyboards having caps-lock and num-lock lights usually light these lights during reset.) The $\overline{\text{RESET}}$ signal is also sent to the keyboard, and at the same time clears U24, a quad flip-flop IC. Meanwhile, the HUMBUG monitor program clears U23a by reading the keyboard (using I/O7 from the address decoder, Fig. 10-1). With U23a cleared, its Q output is low; this signal goes to $\overline{G}$, the enable pin of U25, which allows it to operate. It also goes to U32a pin 1, a 7406 open collector inverter, which then open-circuits its output and lets the keyboard's DATA line swing high or low, as needed. At the same time, the $\overline{Q}$ output of U23a, which is high, goes to the IP2 input of DUART 1.

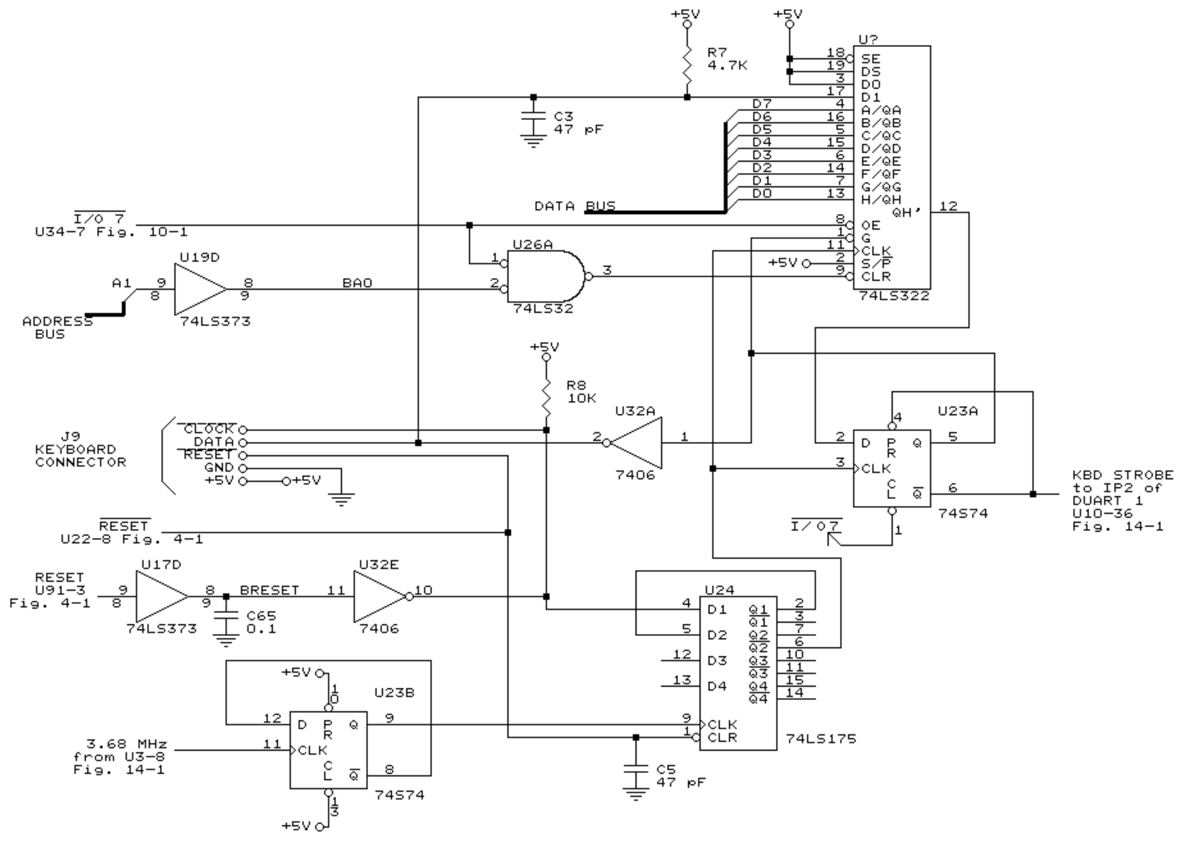


Fig. 15-1. PC-compatible keyboard interface.

Meanwhile, the 3.6864 MHz output of U3 (which is generated for the DUARTs) also goes to U23b, where it is divided by 2 to produce a 1.8432 MHz signal which clocks U24, which is wired as a shift register.

Now suppose you press (or release) one of the keyboard keys. The keyboard sends out a key number (not an ASCII code) on the DATA line as a serial code, and simultaneously starts to pulse the CLOCK line, once for each bit. The data is sent to the D1 input on U25, a 74LS322 shift register. At the same time, the CLOCK signal sends a low pulse to the 1D input of U24, whose first two flip-flops are also wired as a shift register. After two pulses of the 1.8432 MHz clock (a delay of about 1 microsecond), a high comes out the 2Q output and clocks both the U23a flip-flop and U25, the shift register. The data bit then enters the first stage of the shift register, and a moment later the CLOCK signal goes back high.

This process is repeated once for each keyboard data bit, with the data bits proceeding through U25, from QA to QB, QC, and so on, until the first bit gets to the last flip-flop, at which time a high comes out QH' and is sent to the D input of U23a. This bit happens always to be a 1, but it is immediately discarded because at the next clock pulse it is shifted right out of U25. It is there long enough, however, to make U23a set at the next clock pulse. When this occurs, the flip-flop does four things: (a) its $\overline{Q}$ output goes low and sends an interrupt request to DUART 1, which relays it to the 68000, (b) the same low forces U23a to stay set in case there are more clock pulses, (c) the Q output sends a high to the $\overline{G}$ input of U25, which prevents it from shifting further, and (d) the same high is inverted by U32a into a low, which grounds the DATA line and prevents the keyboard from sending further data.

Assuming that DUART 1 is properly programmed, the interrupt request goes to the 68000, which then stops its normal processing and goes to a special routine called an *interrupt service routine* to accept the input from the keyboard. This routine is part of the HUMBUG monitor, which now reads a byte from location $FE01C3; doing this read pulses the $\overline{I/O7}$ line. This clears the U23a flip-flop, and also pulses the output enable ($\overline{OE}$) pin of U25, which then sends the received data byte to the data bus to be read by the 68000. The 68000 thus gets the data byte output by the keyboard.

A few microseconds later, the interrupt service routine does a read from location $FE01C1, which pulses the $\overline{I/O7}$ line a second time, but with a difference. On the first read (from $FE01C3), A1 (and BA0, which is the same as A1 but buffered by U19) was a 1, but on the second read (from $FE01C1) this signal is a 0 (because C3 ends with 0011 whereas C1 ends with 0001; the second bit from the right is A1, which is now a zero.) Hence on the second read, U26a receives two low inputs at the same time, and therefore outputs a low to the $\overline{CLR}$ input of U25, which resets it and gets it ready for the next keyboard output.

Rather than providing a specific ASCII character for each key pressed, PC-compatible keyboards send out a code each time *any* key is pressed, and a different code each time *any* key is released. Since this circuit generates an interrupt request for each such code, the interrupt service routine in HUMBUG is called each time a key is pressed or released, and has the job of keeping track of keys as well as converting these codes into ASCII. A number of books on the IBM PC describe how such a keyboard works; a particularly readable one is chapter 6 of the *Programmer's Guide to the IBM PC* by Peter Norton.

15-2. Construction

Now install the following components:

R7	4.7K 1/4-watt resistor
R8	10K 1/4-watt
U23	74S74 dual D flip-flop and its socket
U24	74LS175 quad latch and its socket
U25	74LS322 shift register and its socket
U17	74LS373 buffer and its socket
U19	74S373 buffer and its socket
C8, C9, and C10	0.1 µF disk capacitors
J9	round, black keyboard connector

U26, U32, and the three 47 pf capacitors have already been previously installed.

15-3. Testing

Now turn on the power. If you have a serial terminal connected, then testing the keyboard circuit is easy:

1. Set the terminal to 2400 baud (the default rate for HUMBUG),
2. Turn on the power,
3. Wait for the beep-boop from the speaker,
4. Press the RETURN key on the PC-compatible keyboard,
5. Press control-S, followed by the letter R, on the PC-compatible keyboard.

You can now use this keyboard for input, but will see all output on the serial terminal.

If your terminal does not run at 2400 baud, you can change the baud rate from the PC-compatible keyboard by inserting the following steps between steps 4. and 5. above:

4a. Type the following exactly as shown: MSFE0003
4b. Press the space bar once
4c. Type in the baud rate code listed in the table in Chapter 14 (but do not type in the dollar-sign).

For example, to change the baud rate to 9600 baud, type in MSFE0003spaceBB (press the space key - don't type in the word "space".) This sets the baud rate register of the DUART (at location $FE0003) to the code $BB, which corresponds to 9600 baud.

Without a serial terminal, you have two choices: either use an oscilloscope or logic probe to check the various signals in the circuit, or else go on to Chapter 16, and test the keyboard after a video board is installed. The latter is probably easier.

Chapter 16

PC-Compatible Bus Connectors

The left rear corner of the SK68K board holds up to six 62-pin card-edge connectors like those of an IBM PC/XT or clone computer. These connectors can hold most PC- or XT-compatible plug-in I/O cards (although cards intended for AT-type slots will not work.)

16-1. Discussion

SK68K software supports the monochrome or CGA color video boards, and the WDXT-GEN (and similar) hard disk controller, but it is fairly straightforward to write software for other cards as well. It does not support some of the other popular PC-type cards such as floppy disk controllers, serial and parallel I/O cards, clock/calendar boards, or multi-function boards which combine several of the above, for the simple reason that all of these options are already contained on the main SK68K board itself and so there is no need for these extra boards.

Although there are many hard disk controllers available for PCs and their clones, SK*DOS - the SK68K disk operating system - currently supports only the Western Digital WDXT-GEN controller, and its older versions such as the WD1002-WX1 version controller. This is because PC-compatible hard disk controllers contain a ROM which contains the 8088 code to operate them. The ROM is customized to fit the particular hardware configuration of the controller. On a true PC or clone, MS-DOS or PC-DOS simply calls the ROM to do the work without having to concern itself with the actual hardware.

But that ROM is in 8088 code; the 68000 in the SK68K can *read* that ROM, but it cannot *understand* it - 8088 machine language has no meaning to the 68000. Thus the disk software in SK*DOS has to do all the work of handling the actual controller hardware, and has to be customized to work with the

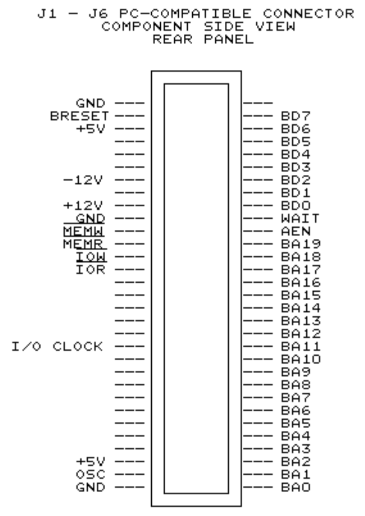


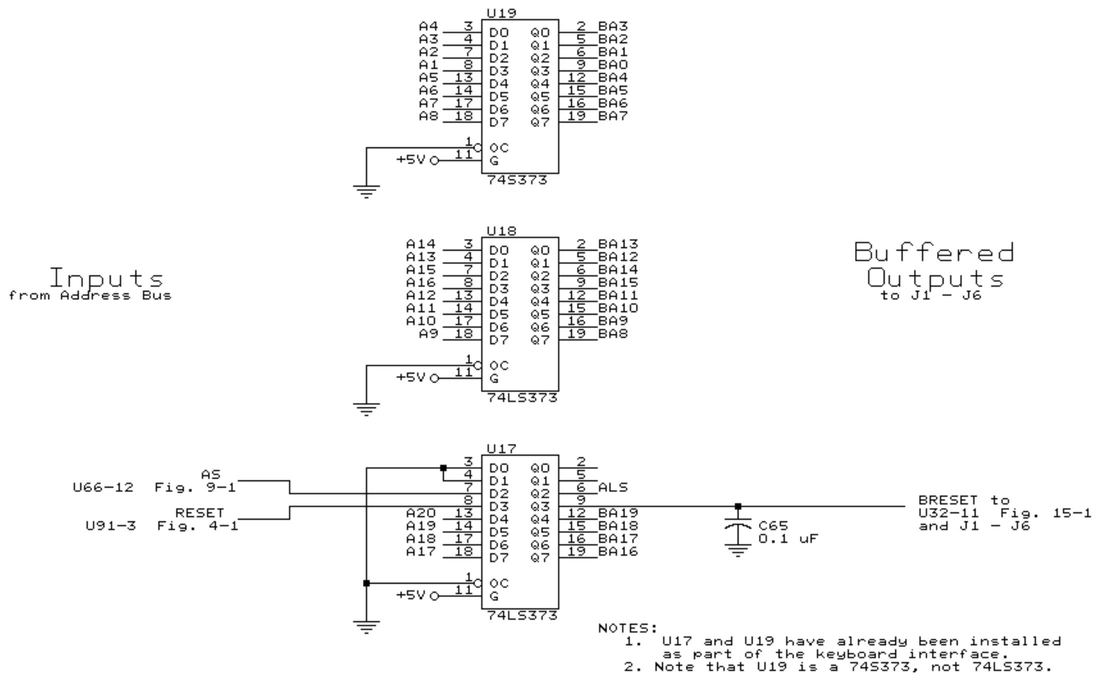
Fig. 16-1. PC-compatible expansion connector.

particular circuitry on a given controller. We therefore chose an inexpensive, widely available hard disk controller and standardized on it.

Fig. 16-1 shows the pinout of each of the six PC-compatible connectors. (This is a top view of the connector, as seen from the front of the board). All of the labelled pins are used; the unlabelled pins are not needed in a 68000 system. Some of the pins, such as ground or power, are obvious; the other connections are shown in the next few diagrams.

Most of the pins on the right hand side of the connector are simply buffered address or data lines. Fig. 16-2 shows the buffering of the address lines, and the data buffers are shown on Fig. 16-4. As you can see in Fig. 16-4, the eight-bit data bus (BD0 through BD7) on the expansion connectors comes from the lower eight bits of the 68000's data bus, so each address on the expansion bus becomes an odd address for the 68000. Moreover, we see that the 68000's A1 line becomes BA0 on the expansion bus, A2 becomes BA1, and so on, up to A20 which becomes BA19. This is necessary because the expansion connectors need a BA0 line, but the 68000 does not have an A0 output; hence everything is shifted over one bit.

To understand how expansion slot addresses equate to 68000 addresses, we have to understand the circuit in Fig. 16-3. In a 68000 (or, in fact, most Motorola processors) there is only one set of addresses, which are used for both memory and I/O. The 8080 (and most Intel processors) have two sets of addresses - one for memory, and another just for I/O. Memory addresses



NOTES:
1. U17 and U19 have already been installed as part of the keyboard interface.
2. Note that U19 is a 74S373, not 74LS373.

Fig. 16-2. Address bus buffers.

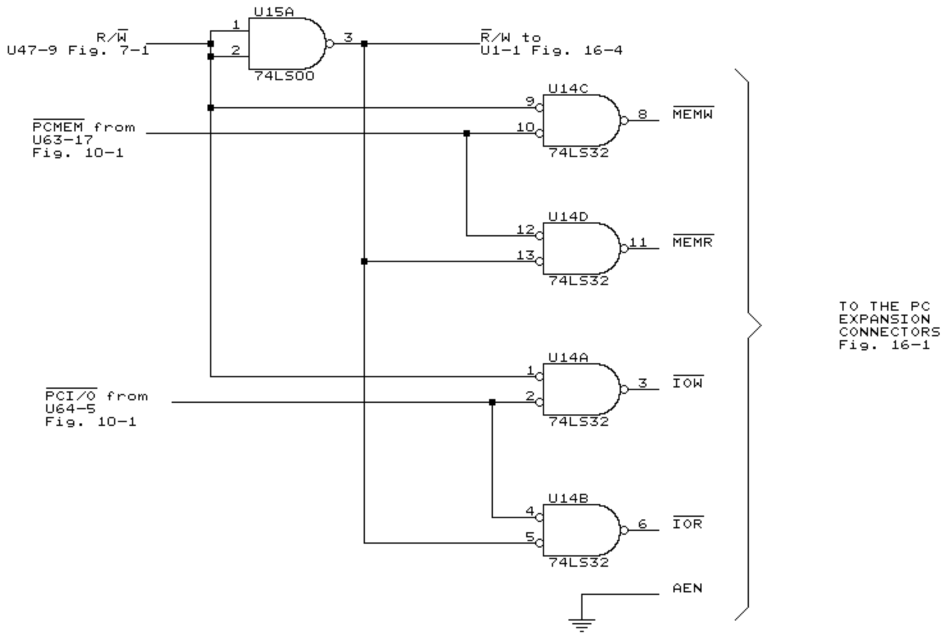


Fig. 16-3. Memory and I/O select logic.

are used for normal reads and writes, while I/O addresses are used only by special IN and OUT instructions. A typical Intel processor then manipulates these addresses with four control lines:

MEMW is asserted (low) to write to memory
MEMR is asserted to read from memory
IOW is asserted to write to an I/O device
IOR is asserted to read from an I/O device.

In a typical IBM PC or clone, these four signals are brought to the bus connectors, and all four are often used by I/O cards. For example, on a monochrome or color video board, IOW and IOR are used to control the card, but MEMW and MEMR are used to access the video RAM which stores the data to be displayed. On a hard disk controller, IOR and IOW again control the hardware, but MEMR is needed to read the ROM on the board.

Although most PCs or clones have a maximum of 640K of memory, they can actually address 1 megabyte of memory and 64K of I/O addresses, using all 20 address bits for memory and 16 bits for I/O addresses. The megabyte of memory addresses has room for 640K of plain RAM, plus 128K of video memory, 64K of hard disk ROM, and up to 192K of other ROM such as the BIOS and ROM Basic. All of these addresses - both memory and I/O addresses - have to be squeezed into the 68000's single memory address space.

As we saw in Fig 10-1, the address decoder generates a PCMEM signal for 68000 addresses \$C00000 through \$DFFFFF, and a PCI/O signal for addresses \$FA0000 through \$FBFFFF. These two enable signals are used by Fig. 16-3 as follows:

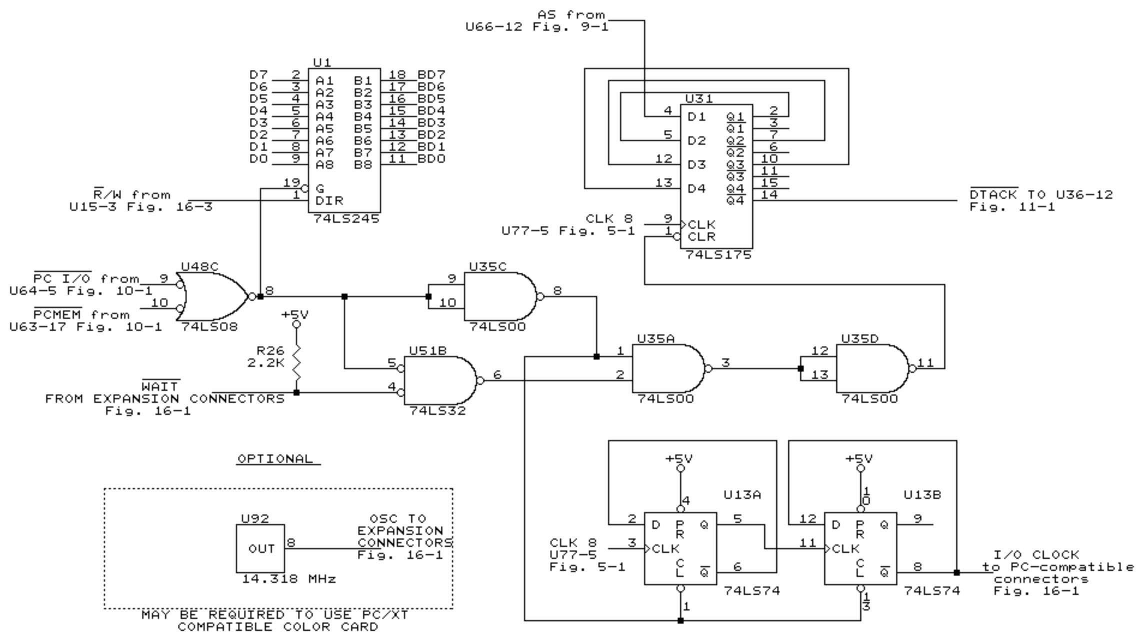

AS from U66-12 Fig. 9-1
U1
74LS245
R/W from U15-3 Fig. 16-3
U31
74LS175
DTACK TO U36-12 Fig. 11-1
CLK 8 U77-5 Fig. 5-1
PC I/O from U64-5 Fig. 10-1
PCMEM from U63-17 Fig. 10-1
U48C
74LS08
U35C
74LS00
+5V
R26 2.2K
WAIT FROM EXPANSION CONNECTORS Fig. 16-1
U51B
74LS32
U35A
74LS00
U35D
74LS00
OPTIONAL
U92
OUT
14.318 MHz
OSC TO EXPANSION CONNECTORS Fig. 16-1
MAY BE REQUIRED TO USE PC/XT COMPATIBLE COLOR CARD
U13A
U13B
74LS74
CLK 8 U77-5 Fig. 5-1
I/O CLOCK to PC-compatible connectors Fig. 16-1

Fig. 16-4. Miscellaneous expansion circuitry.

1. If $\overline{\text{PCMEM}}$ is low and R/$\overline{\text{W}}$ is low, then U14c generates a low $\overline{\text{MEMW}}$ signal.
2. If $\overline{\text{PCMEM}}$ is low and R/$\overline{\text{W}}$ is high, then U14d generates a low $\overline{\text{MEMR}}$ signal.
3. If $\overline{\text{PCI/O}}$ is low and R/$\overline{\text{W}}$ is low, then U14a generates a low $\overline{\text{IOW}}$ signal.
4. If PCI/O is low and R/W is high, then U14b generates a low IOR signal.

Thus when the 68000 reads or writes into memory addresses $C00000 through $DFFFFF, cards plugged into the expansion slots get a memory read or memory write signal; when the 68000 writes into memory addresses $FA0000 through $FBFFFF, the cards get an I/O read or I/O write signal.

The result is that cards in the expansion connectors can be written to or read, but the address they get is slightly different from the address the 68000 is accessing. For PC memory addresses, the relationship is this:

PC memory address	68000 memory address
$00000	$C00001
$00001	$C00003
$00002	$C00005
$00003	$C00007
:	:
$FFFFF	$DFFFFF

We can use the formula

68000 memory address = $C00001 + 2 x (PC memory address)

to convert one to the other. For example, the top left corner of a monochrome video board is at address $B0000, which translates to address $D60001 in the SK68K.

For PC I/O addresses, the relationship is this:

PC I/O address	68000 memory address
$0000	$FA0001
$0001	$FA0003
$0002	$FA0005
$0003	$FA0007
:	:
$FFFF	$FBFFFF

We can use the formula

68000 memory address = $FA0001 + 2 x (PC I/O address)

to convert one to the other. For example, the control port of a monochrome video board is at I/O address $03B8, which translates to address $FA0771 in the SK68K.

The final result is that the 1 megabyte of PC RAM translates to 2 megabytes of 68000 addresses, from $C00000 through $DFFFFF, while the 64K of PC I/O addresses translate to 128K of 68000 addresses, from $FA0000 through $FBFFFF, but only the odd addresses are used in the SK68K because only the lower half of the 16-bit address bus is connected to the expansion connectors. This means that it is not practical to connect a PC-type memory board to the SK68K since the board could only store odd addresses.

The rest of the circuitry is shown in Fig. 16-4. Whenever either $\overline{\text{PCI/O}}$ or $\overline{\text{PCMEM}}$ goes low, indicating that the 68000 is trying to access the expansion connectors, U48c outputs a low $\overline{\text{PCEN}}$ signal which enables U1, the bidirectional transceiver which buffers the BD0 through BD7 data lines to the expansion connectors; the direction of data flow is determined by the $\overline{\text{R}}$/W signal (which is just R/$\overline{\text{W}}$ inverted; this signal is therefore low when reading and high when writing.)

PCEN also goes to U35c, which inverts it to a high PCEN signal; this permits U13a and U13b to divide the 8 MHz CLK8 signal by 4 and send the 2 MHz I/O CLOCK to the connectors. This signal is used by some video boards as a clock for an MC6845 video controller chip.

PCEN also goes through U35a and U35d to the clear input of U31. This IC is wired as a shift register to produce a time delay. In normal operation, U31 is held cleared and does nothing. But when PCEN goes high, U31's clear input also goes high and it starts to shift a high (from AS) through the register, one flip-flop for every pulse of CLK8. After four clock pulses, or about 500 nanoseconds, the 4Q output goes low and sends $\overline{\text{DTACK}}$ to U36.

This gives PC-compatible cards 500 nsec to work, but some cards need additional time and send back a low PC/XT $\overline{\text{WAIT}}$ signal. This signal goes through U51b and prevents U31 from timing out until the $\overline{\text{WAIT}}$ signal goes high. U31 then gives these cards an extra 500 nsec or so after the $\overline{\text{WAIT}}$ signal returned to high.

Finally, Fig. 16-4 also shows the 14.31818 MHz oscillator; it is needed by some color video boards. (This signal is four times 3.579545 MHz, which is the color burst frequency). Some color boards have their own oscillator; others need one on the motherboard; either way, it is convenient to supply this signal in all cases.

16-2. Construction

Now install the following components:

J1 through J6	62-pin card edge connectors (if you install fewer than six, then space them apart)
U18	74LS373 and its socket
U15	74LS00 and its socket
U14	74LS32 and its socket
U1	74LS245 and its socket
U48	74LS08 and its socket
U35	74LS00 and its socket
U51	74LS32 and its socket
U31	74LS175 and its socket
U13	74LS74 and its socket
U92	14.31818 MHz oscillator (soldered directly to the board, and with the pointed corner identifying pin 1 closest to J4)
R26	2.2K 1/4-watt resistor
C1	0.1 µF disk ceramic capacitor

U17 and U19 have already been installed.

Finally, place a short wire jumper from U15 pin 12 to pin 7. Now that we have installed U15, part of that IC is generating a false $\overline{\text{DTACK}}$ which is upsetting everything else. For now, this wire jumper disables this circuit; we will remove the jumper as soon as we install U16 in a future step.

16-3. Testing

If you have a PC-compatible video board and matching monitor, plug it in at this time and turn on the power. If all goes well, on the screen you should now see the message "Please press enter".

Some users have reported problems with a CGA (color graphics) board being unreliable. This is often due to noise on the RESET line at the PC slots, which resets the board when it shouldn't. If you encounter this, place a 0.1 µF disk capacitor right on the CGA board, from pin B2 (left side of connector, second pin from rear - see sheet 4 of your diagram) to ground (which is on pin B1, about 1/4 inch away). Don't solder to the plated pin - find a solder pad a few tenths of an inch away.

In case of difficulty, follow the general procedures in section 13-4 of this manual. To make the process more understandable, let's first discuss what should happen when all is working correctly, and how the computer decides which I/O devices to use.

When you turn on the power (or short the reset pins at J23), the HUMBUG monitor program in the computer's ROM tries to initialize the input and output ports, makes a list of what options you have installed, and sounds the beep-boop from the speaker. If it detects that a video board is plugged into one of the interface connectors, it will then display a "Please press Enter" in the top left corner of its monitor; it does not, however, display that message on a serial terminal because it doesn't yet know what baud rate to use.

HUMBUG is now monitoring both the serial input and the keyboard connector, waiting for you to press the ENTER key (also called RETURN or CR), so it can determine (a) which keyboard you will be using, and (b) what baud rate you are using if you choose the serial keyboard. Both keyboards can thus be connected, but the first one to get an ENTER will be chosen as the input device. (In order to make sure the correct baud rate is chosen, you may have to press ENTER several times on a serial keyboard.)

After the ENTER is received, HUMBUG displays its sign-on message, the prompt (*), and a cursor (an underline in this example):

```
HUMBUG (R) Copyright (C) 1986-1991 by Peter A. Stark
*_
```

Whichever keyboard you use, this message will go to the video board(s), if any. If you use a serial keyboard, then it will also go out the serial port to the terminal (at the same baud rate as the keyboard). Once you get the * prompt and cursor, you may type in any of 32 HUMBUG commands; we will get to those later. For now, try typing in the command `HE` to get a Help screen which shows the HUMBUG commands.

Chapter 17

Computer Memories

To best understand how the dynamic RAM circuitry of our computer works, let us make a short detour to look at memories in general. (Construction will continue in Chapter 18.)

17-1. Memory Basics

Let us begin with a look at what is inside a simple memory IC, shown in Fig. 17-1. The heart of such an IC is the *memory array* which contains the actual *memory cells* (the cells store the actual memory data.). The array shown in Fig. 17-1 consists of just four horizontal wires called *rows* and four vertical wires called *columns*, whereas a real memory chip will often contain hundreds of rows and columns. At the intersection of each row and column is a cell. Since we have four rows and four columns, the array shown has room for exactly 16 (4 times 4) cells. (Notice that the row and column wires, though they are shown as crossing, do not actually connect to each other. Instead, there is a cell at each intersection, and that cell has one connection to the row wire and another connection to the column wire.)

To accommodate 16 cells, we could have used one row and 16 columns, or 2 rows and 8 columns, or several other combinations, but in practice memory arrays generally have the same number of rows and columns because that makes the rest of the circuitry simpler. In other words, memory arrays generally tend to look like a square rather than a rectangle. Furthermore, the number of rows and columns is always a power of 2. For example, a 16K memory chip would have 128 rows and 128 columns, for a total of 128x128 or 16,384 cells. The next larger common memory chip would have 256 rows and 256 columns, for a total of 256x256 or 65,536 cells. This explains why 16K and 64K memory chips are common, but 32K chips are not - their array would be a rectangle rather than a square.

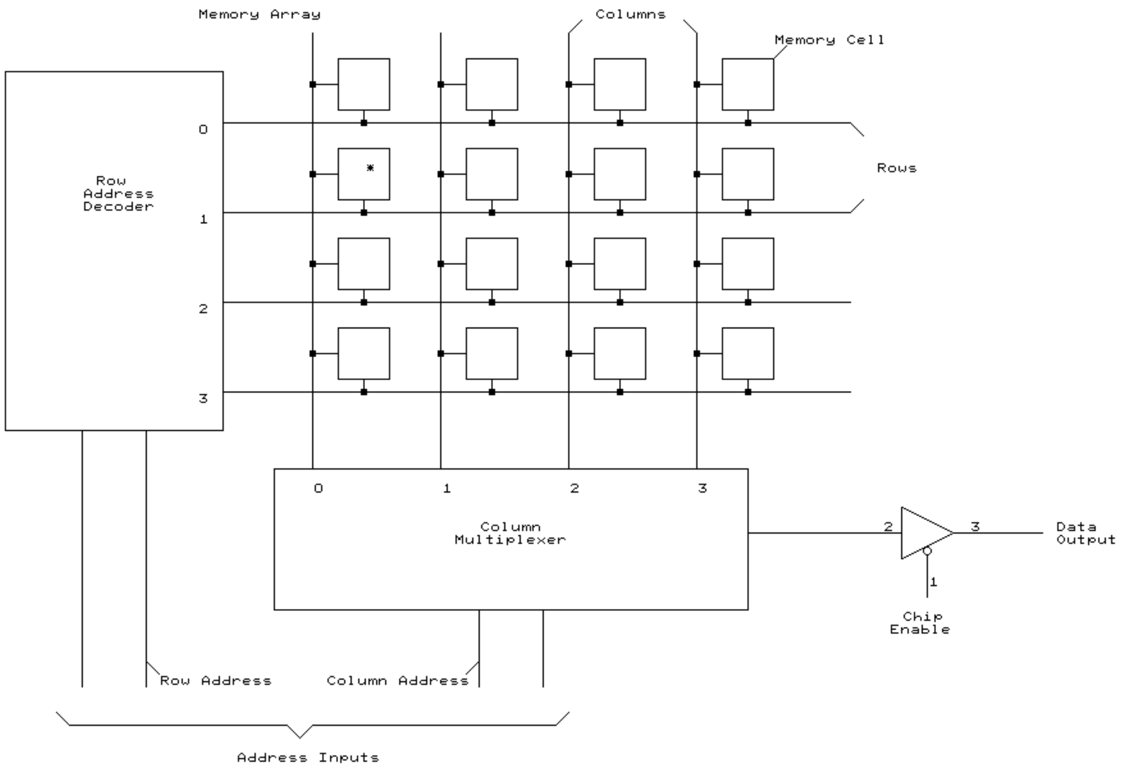


Fig. 17-1. A simple memory IC.

Each cell in the array stores one bit, so the circuit of Fig. 17-1 can store 16 bits. This particular circuit is configured to have 16 locations, each of which stores one bit; such an IC would be called a 16x1 memory, where the first number gives the number of locations while the second gives the number of bits in each location. Small memory ICs often have more than one bit per location, whereas large memory ICs almost always have just one bit in each location, but with many thousands of locations.

Each of the locations in the memory (that is, each cell) has an *address*; each time we want to read from or write into a cell (though Fig. 17-1 doesn't show the circuitry needed to write into a cell), we must specify the address of the specific cell to read or write by giving the IC a binary address on the address input lines. Since this circuit has 16 cells or locations, it requires a 4-bit address to specify the exact cell we want to access. The general rule is that x address bits are needed to specify 2^x addresses, so 4 address bits specify 2^4 or 16 locations in our simple circuit.

The four-bit address is split into two parts - a two-bit row address and a two-bit column address. Since there are four rows, we need two bits to choose one of them (again, because 2^2 is 4); since there are four columns, we need two bits to choose a column. Keeping in mind that most memory ICs have square arrays - the same number of rows and columns - that means that they will need the same number of row address bits as column address bits. This means that the *total* number of address bits is almost always an even number. For example, a 16Kx1 IC with 128 rows and 128 columns has 7 row address bits and 7 column address bits (since 2^7 is 128), for a total of 14 address bits (and 2^{14} is 16,384). Similarly, a 64Kx1 IC has 256 rows and columns, 8 row and column address bits (2^8 is 256) and a total of 16 address bits (2^{16} is 65,536 or 64K).

Let's suppose that we want to read out the bit in location 4 of the circuit in Fig. 17-1, represented by the starred cell in the diagram. To do so, we send the binary address 0100 (a four) to the address inputs. The left two bits, 01, become the row address, while the right two bits, 00, become the column address. The row address is sent to a *row address decoder*, which has two inputs (for the row address) and four outputs (labelled 0, 1, 2, and 3) which correspond to rows 0, 1, 2, and 3. A decoder normally has a number of outputs, all of which are off except for one - the one specified by the binary input. In this case, the binary input (the row address) is 01, so the decoder turns on its 1 output and turns off the 0, 2, and 3 outputs.

Of the 16 cells in the array, the 12 cells which are in rows 0, 2, and 3 receive no signal from the decoder, so they do nothing. But the four cells in row 1 all receive a signal from the number 1 output of the decoder, so they all get enabled. In turn, each of these four cells sends its bit down a column wire to the *column multiplexer*. In other words, although we only want the contents of *one* cell - the starred one - all four cells along the same row send their contents down to the multiplexer.

The multiplexer's job is now to select the *one* desired bit from the four it receives and send it out the output. To do so, it acts like a SP4T switch - a single-pole switch with four positions, which selects one of the four inputs and sends it out the output. The precise input selected depends on the binary column address - in our case, the column address is 00 so it selects the signal entering on the input labelled 0 and sends it out to a tri-state buffer. If the chip enable input is on (it has to be low, since the tri-state buffer has an active-low enable input as shown by the bubble), then that bit goes out the data output.

As mentioned earlier, the circuit of Fig. 17-1 is very simplified. In an actual memory IC, the chip enable signal might also go to the decoder or multiplexer to prevent their working unless the chip is selected, there would be circuitry to write into cells (along with a R/W input), the array would be much larger, and there would be more components there that we haven't yet discussed.

The next question, though, is this - what is in a cell? That depends on the type of memory IC we are discussing. In a ROM or PROM (a Programmable ROM), the cell might consist of just a diode, or a diode in series with a fuse. Such a cell can store either a 0 or 1 bit, depending on whether the diode is connected or not (for example, if the series fuse is blown out). In an EPROM, the cell consists essentially of a FET transistor which is biased on or off by a charge stored in an insulating region.

In RAMs, there are two main kinds of cells: in a static RAM (or SRAM), the cell consists of a flip-flop which stores a 0 or 1, depending on whether it is set or reset, plus some additional components which connect the flip-flop to the row and column wires. Since this involves several transistors (often six or more in each cell), the static RAM cell is quite complex. A DRAM cell, on the other hand, consists of just one MOSFET transistor and a tiny capacitor, which stores a 0 or 1 depending on whether it is charged or not. Since the DRAM cell is so much simpler than a SRAM cell, DRAM ICs generally contain many more cells than SRAM ICs. On the other hand, DRAM memories require more external support circuitry than SRAM, and are somewhat slower. Thus smaller memories are usually made of static RAM chips, whereas larger memories are generally dynamic RAMs (except in those cases where absolute top speed is a necessity and cost is no object.) In the SK68K computer, for example, there is a small amount of static memory (consisting of just two ICs) which allows us to get the system up and running quite quickly. But the main memory, 1 megabyte worth, is strictly dynamic to keep the total cost down. Even though the DRAM needs extra support circuitry to make it work, the circuitry is shared by the entire 1 megabyte of RAM so it is worth using, whereas it would not be worth using if only a few K of memory were needed.

17-2. Dynamic Memory (DRAM)

Fig. 17-2 shows a typical DRAM cell, consisting of a storage capacitor which holds the actual bit, in series with a MOSFET transistor. In normal operation, the decoder output is off and so the MOSFET transistor is biased off. This isolates the storage capacitor from the rest of the circuit so it can hold a bit. But when the row holding the cell is selected by the decoder, the MOSFET is biased on and the capacitor is connected to the column wire through the transistor. At this point, the cell can be read out (since the capacitor voltage appears on the column wire) or it can be written into (by sending a signal up the column wire, and through the MOSFET transistor into the capacitor.)

If this were all there was, there would be two major problems: First, since the capacitor is *very, very* small, it discharges very quickly. Just reading the cell (by turning on the MOSFET) puts enough of a load on the capacitor that it discharges almost instantaneously, but even when the MOSFET is biased off, the capacitor will typically hold its charge only a few seconds, and under some conditions, only a few milliseconds. Moreover, whenever a row wire is turned on by the decoder to select a cell on that row, **all of the MOSFETs on that row are turned on!** In other words, reading one cell selects all the cells on that row, with the result that all the capacitors in that row immediately discharge. Thus something has to be done to prevent all

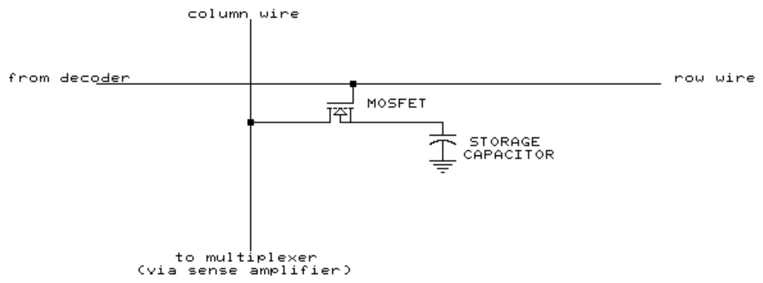


Fig. 17-2. A dynamic RAM cell.

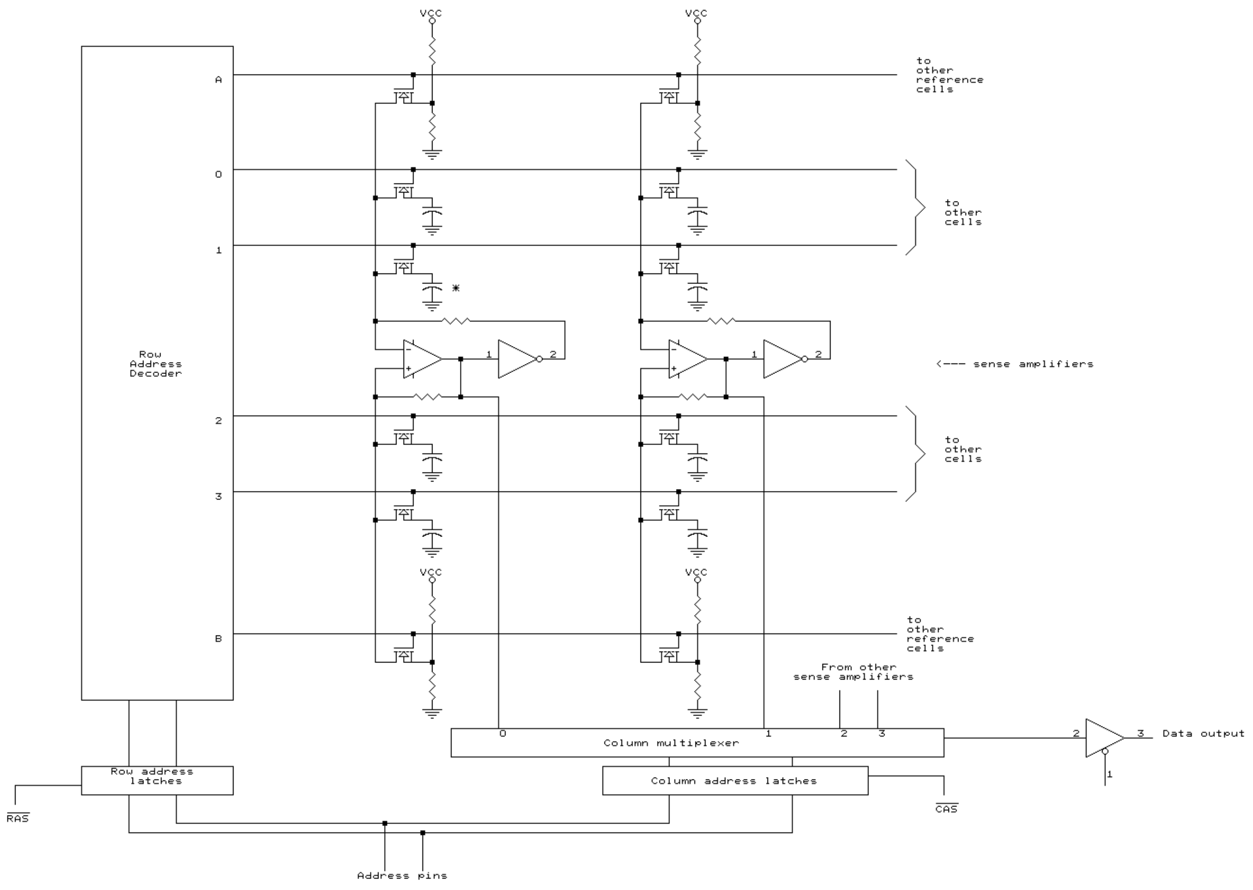


Fig. 17-3. DRAM IC organization.

the cells from forgetting their data. The overall dynamic RAM system therefore has additional circuitry which will (1) rewrite all the data back into all the cells in a row whenever any cell in that row is read or written, and (2) rewrite the data into all the cells of the entire memory at intervals of a few milliseconds. The first job - that of rewriting cells when a row is accessed - is handled internally by each DRAM IC; the second job - that of rewriting all of memory every few milliseconds - is called *refreshing* and is handled by external *refresh circuits*.

Fig. 17-3 shows how the circuitry inside a typical DRAM integrated circuit handles rewriting data into the cell capacitors whenever a row is selected. Though DRAMs typically contain thousands of cells, this diagram still shows only a small 16x1 chip with an array of four rows and four columns, though only the left two columns are actually shown in the diagram.

Fig. 17-3 has several new components we have not seen before. Although the 16x1 DRAM circuit needs four address bits, the diagram shows only two address pins, connected to both the row address decoder and the column multiplexer through two sets of flip-flops called the *row address latches* and the *column address latches*. The four-bit address is sent to the IC two bits at a time; the first two bits are stored in the row address latches by a pulse on the $\overline{\text{RAS}}$ or *row address strobe*, and then the second two bits are stored in the column latches by a pulse on the $\overline{\text{CAS}}$ or *column address strobe*. This technique is used in DRAM ICs to save input pins. For example, a 256Kx1 DRAM would normally need 18 address lines, whereas splitting the address into two parts allows the 18 bits to be input through 9 address pins and two address strobe pins. This allows 11 pins to do the work of 18, so the DRAM can be packaged in a smaller case. There is, of course, a disadvantage - external circuits have to be added to split the address into two parts, and the process takes a bit longer than it would otherwise.

As before, the two-bit row address is again sent to the row address decoder, which outputs a pulse on one of the four row lines labelled 0, 1, 2, or 3, depending on the value of the row address. Note, however, that the array is now split into two parts by a row of *sense amplifiers* running across the middle of the diagram. These sense amplifiers are essentially op-amps or comparators, and have an inverting input (labelled with a -) as well as a non-inverting input (labelled with a +).

But now there are two more rows, labelled A and B, each of which contains a row of *reference cells*. These cells are similar to regular cells, except that they use a two-resistor voltage divider instead of a capacitor and therefore always output a constant voltage whose value, chosen by the ratio of the two resistors, is half-way between a 0 and a 1. The decoder is now modified so that, in addition to selecting one of the rows 0 through 3, it also selects a reference row at the same time, but it always makes sure that the reference row is on the opposite side of the sense amplifiers.

For example, let's again assume we want to read out the contents of the starred capacitor, which is again location 4 or 0100. The row address (01) entering the decoder selects row 1; at the same time, the decoder enables the row of reference cells connected to output B. Note that the reference row is on the opposite side of the sense amplifiers - row B gets selected along with rows 0 or 1, whereas row A gets selected along with rows 2 or 3.

When row 1 is selected, all of the MOSFET transistors connected to that row are turned on, and all four capacitors on that row (only two are actually shown) send their voltage to the top input to a sense amplifier. At the same time, the bottom input of each sense amplifier gets a reference voltage from a reference cell at the bottom. Since the reference is a voltage between 0 and 1, each sense amplifier can compare the capacitor voltage against the reference and decide whether the capacitor held a 0 or 1.

All of this must happen very quickly, because the capacitor almost immediately discharges. By then, however, the bit stored in the capacitor has already arrived at the output of the sense amplifier, which sends it down to the multiplexer and the output.

But each of the sense amplifiers has a sort of positive feedback circuit connected from output back to the two inputs. The actual circuit is somewhat different and more complicated than what is shown in Fig. 17-3, but the main idea is this: as soon as the bit arrives at the output of the sense amplifier, it is immediately sent back to the inputs through the two resistors. Since the top sense amplifier input is inverting, the output is fed back through an inverter so it comes back in the same polarity, but larger. This signal then recharges (refreshes) the capacitor. if the original capacitor voltage was lower than the reference voltage, this circuit pushes it back down toward ground; if it was higher than the reference, this circuit pushes it up toward the positive supply voltage.

Even though we only wanted to read out the bit stored in the starred capacitor, actually every capacitor in row 1 was read out and refreshed by its own sense amplifier. This is an important concept - reading out any cell automatically refreshes all the cells located in the same row.

If such a DRAM IC is connected to a microprocessor, each time the processor reads (or writes - Fig 17-3 does not show any of the circuitry for writing into a cell, but this can be accomplished by feeding bits backward through the multiplexer) it refreshes an entire row of each DRAM IC. If the computer were to use data from every row, then the entire IC would be refreshed automatically and we wouldn't have to do any more. In general, though, we cannot trust that to happen, since the computer could easily get stuck in a loop where it only accesses one or two rows, with the result that all the rest of the memory would be forgotten. We therefore have to add external refresh circuitry to make sure that every row of each DRAM IC is properly refreshed.

17-3. DRAM Refreshing

Most current DRAM ICs require that they be completely refreshed once every 2 milliseconds, so the refresh circuits have to ensure that every row of the memory is accessed at least once every 2 milliseconds. To minimize the effort required, IC makers build larger DRAM chips a bit differently from the smaller chips. In smaller DRAMs, up to 16Kx1, the array is essentially square as we have discussed, and has 128 or fewer rows. In larger memories, the array wiring is split to make several smaller arrays out of the one large array, with each smaller array having only 128 rows and all arrays being refreshed at the same time. As a result, instead of a 256Kx1 DRAM having 512 rows and thus needing 512 reads for refreshing, it still only needs 128 reads. This makes refreshing faster.

There are two basically different ways of refreshing DRAM:

(1) The cheapest, requiring very little actual hardware, is to build an oscillator which interrupts the CPU once every 2 milliseconds., and forces it to stop the current program and execute an interrupt routine. The interrupt routine, in turn, does a read from every row and then returns to the main program. This is essentially a software approach, but it has the disadvantage that it wastes a significant portion of the computer's time and slows down every program. For instance, if a read of one row (counting the time to fetch and perform the read instruction) requires four microseconds (which is not unusual for an average 8-bit microprocessor), then 512 microseconds (plus interrupt processing time) would be taken up out of every 2 milliseconds for refresh. In other words, more than a quarter of the processor's time would be used up just on refresh.

(2) The second approach is to build a counter which counts out the rows from 0 through 127, and send its output as a refresh address to the memory. The counter must go through the complete count at least once every 2 milliseconds, and every one of those counts must be sent to the memory. The trick here is to do all this without slowing down the processor or

affecting any programs. As usual, designers use many different ways, some better than others:

(a) One approach is to periodically halt the processor, and send the 128 counts to the DRAMs instead, either individually or as a burst. If the computer has a direct memory access (DMA) circuit, then this is easily handled by the same circuitry. IBM PCs and clones use this approach, but it obviously slows down programs and so is not the best.

(b) The best - and most expensive - approach is to split the memory into two halves. Whenever the CPU is accessing one half, refresh the other half. For example, in an 8-bit computer, all the even locations could be in one half of the memory, and all the odd locations in the other half. Since most of a computer's time is spent accessing consecutive locations, it mostly alternates from one half to the other, so each half of memory is unused roughly 50% of the time. This leaves plenty of time to do memory refresh without in any way slowing down the processor.

(c) A middle-of-the-road approach is to try to detect clock cycles when the CPU is doing internal operations instead of using the memory, and squeeze refresh accesses into these unused slots. With a processor which uses many cycles for internal operations, this can sneak in refreshes without slowing down the processor at all, but this approach doesn't work quite as well with processors which use the address bus fairly heavily. In that case, the refresh circuits may be able to sandwich many - or even most - of their memory accesses between CPU memory accesses, but there may still be occasional conflicts when both need to access memory at the same time. In that case the refresh circuits must get priority to avoid losing data in memory, so there has to be a fairly complex circuit which times refreshes and arbitrates between CPU and memory accesses. This is the approach used in the SK68K computer, and it slows down CPU operation an average of one or two percent.

Chapter 18

DRAM Circuitry

In this chapter we describe and build the actual DRAM circuits in our SK68K computer. Since we have already discussed the general principles behind DRAM circuits, we will discuss the specific DRAM circuitry of the SK68K.

18-1. Discussion

Block Diagram Description

The block diagram of the complete DRAM circuitry is shown in Fig. 18-1. The overall timing of refreshing is handled by the DRAM control circuits, which receive RSHAS and the DRAM enable signal from the address decoder, the AS address strobe, and a clock, and which generate the RAS, CAS, and DTACK signals, as well as three enable signals which control three sets of tri-state buffers.

The three sets of tri-state buffers are separately enabled by the DRAM control circuits, with only one set of buffers enabled at any one time. In this way, three different inputs can be combined onto one set of pins. When the CPU is accessing memory, these buffers would work in this order:

1. The 68000 outputs an address, the address decoder recognizes a DRAM address and sends the DRAM enable signal to the DRAM control circuits.
2. The control circuits enable tri-state buffers A (with B and C disabled) to send nine bits of the address to the DRAM ICs, and then pulse RAS to latch them in the row address buffers within each DRAM.
3. The control circuits then enable tri-state buffers B (with A and C disabled) to send the other nine bits of the address to the DRAM ICs, and pulse CAS to latch them in the column address buffers within the DRAM.

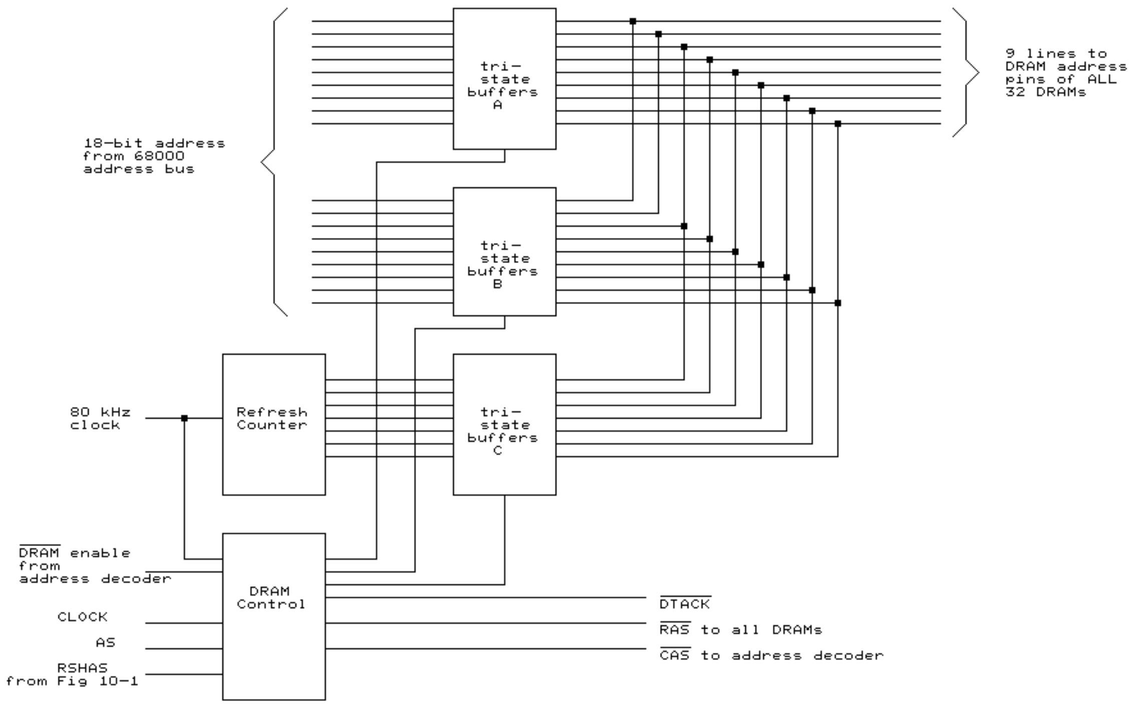

18-bit address from 68000 address bus
tri-state buffers A
tri-state buffers B
tri-state buffers C
9 lines to DRAM address pins of ALL 32 DRAMs
80 kHz clock
Refresh Counter
$\overline{DRAM}$ enable from address decoder
CLOCK
AS
RSHAS from Fig 10-1
DRAM Control
$\overline{DTACK}$
$\overline{RAS}$ to all DRAMs
$\overline{CAS}$ to address decoder

Fig. 18-1. DRAM section block diagram.

The $\overline{RAS}$ signal goes to each DRAM IC so that all ICs, even those not being used, receive a row address. But the $\overline{CAS}$ signal goes back to the address decoder, in Fig. 10-1, which steers the CAS to the appropriate group of ICs.

The completed SK68K computer uses 32 256Kx1 DRAM ICs to provide a total of one megabyte of RAM. These ICs are organized in four banks of 256K bytes each as follows:

U38 through U45 hold all the odd bytes for the first 512K
U53 through U60 hold all the even bytes for the first 512K
U67 through U74 hold all the odd bytes for the second 512K
U80 through U87 hold all the even bytes for the second 512K

When a particular byte (or 16-bit word, in the case of a two-byte transfer) is accessed, only one or two of these banks need be enabled. Thus the address decoder does the final selection, based on the state of A19, $\overline{UDS}$, and $\overline{LDS}$, and sends $\overline{CAS}$ only to those banks being accessed. The DRAM ICs use $\overline{CAS}$ as their main chip enable, and only those ICs getting $\overline{CAS}$ do an actual read or write.

Refreshing is initiated by the 80 kHz clock input to the refresh counter and the DRAM control circuits. Once every 12.5 microseconds, a clock pulse arrives, increments the refresh counter to a new row address, and sends a refresh request to the DRAM control circuits. The control circuits then wait until the 68000 stops using the memory and begin the refresh sequence by enabling tri-state buffers C (with A and B disabled) to send the refresh address to the DRAM chips, followed by a pulse on the RAS line. Since RAS as well as the address lines go to all 32 DRAMs, all the DRAMS are refreshed at the same time.

The complete refresh cycle for all 128 rows takes 1.6 milliseconds (12.5 microseconds between clock pulses times 128 rows), which is somewhat less than the 2 milliseconds specified for most DRAMs, but there is nothing wrong with refreshing the DRAMs more often than necessary.

DRAM Operation During CPU Accesses

Let's begin our discussion with the timing circuits shown in Fig. 18-2, and let's assume flip-flops U49a and U49b are reset, which is the condition most of the time (for U49a is reset every time address strobe AS goes low). Since U49b is reset, its REFRESH output is low and REFRESH is high. Furthermore, since U49a is also reset, the CAS and DRAM DTACK outputs are both high.

When the 68000 wants to access DRAM to either write into it or read from it, it places a valid DRAM address on the address bus. The address decoder (Fig. 10-1) recognizes the address as referring to the DRAM, and sends out the DRAM enable signal which goes to Fig. 18-2 and starts a DRAM cycle. (The signal travels a long distance on the board, and C68, a 33 pF capacitor, is used to reduce its noise pickup.)

The active-low DRAM signal goes low at this point; since pin 10 of U51c is also low as we will see in a moment, U51c sends out a low pulse to U37c pin 10.

At this point it pays to recap for a moment. U51c is actually an OR gate, but it is shown as an AND gate with bubbles on both its inputs and outputs. This notation is used because its job in this circuit is not to *or*, but to *and* two signals. In this case, when both of its inputs go low - and only then - its output goes low. U37c, on the other hand, is a NAND gate but it is shown as an OR gate with bubbles on its inputs. This notation is used because its

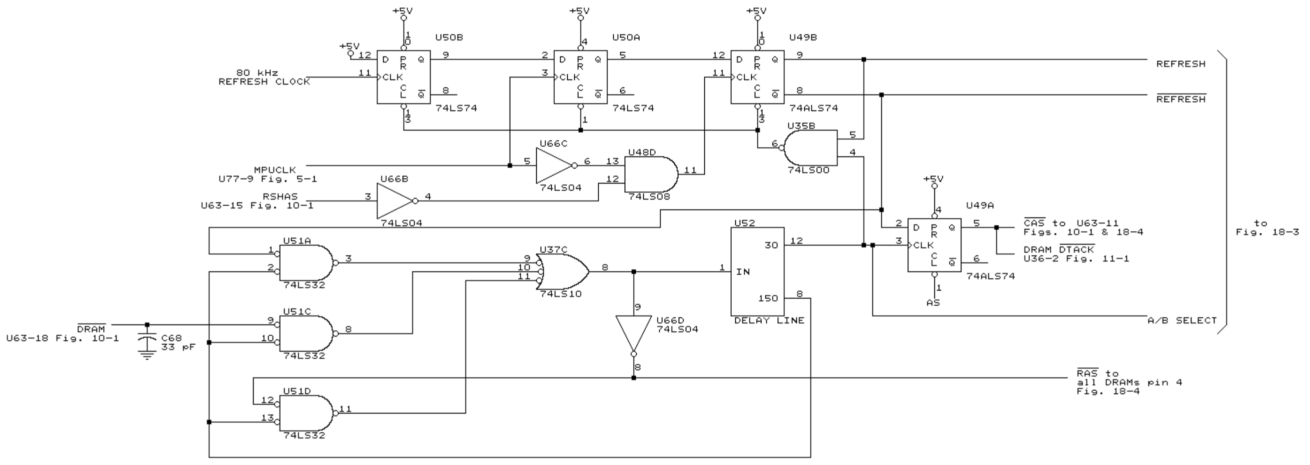


Fig. 18-2. DRAM timing circuits.

job is not to *and* three signals, but to *or* them. Its job is to output a high whenever any one of its inputs goes low. In other words, there are three conditions under which pin 8 of U37c can go high.

As we mentioned two paragraphs ago, when the 68000 wants to access DRAM, it sends out the $\overline{\text{RAM}}$ signal which, in turn, makes U51c pin 8 go low. As a result, U37c pin 8 goes high. This signal now does three things.

First, it is inverted to a low by U66d to generate the $\overline{\text{RAS}}$ signal. This is the *row address strobe* which goes to the DRAM ICs to tell them to accept a row address. The same signal also goes back to U51d pin 12; since pin 13 is already low (just like pin 10), U51d pin 11 also now becomes low. This provides a second low to U37, making sure that it continues to output a high on pin 8 even if the $\overline{\text{RAM}}$ signal should disappear at this point. This is not important right now, though, since the $\overline{\text{RAM}}$ signal is not going to disappear until after the DRAM operation is completed (this circuit is only needed during refreshing). Finally, the high on U37c pin 8 also goes to U52.

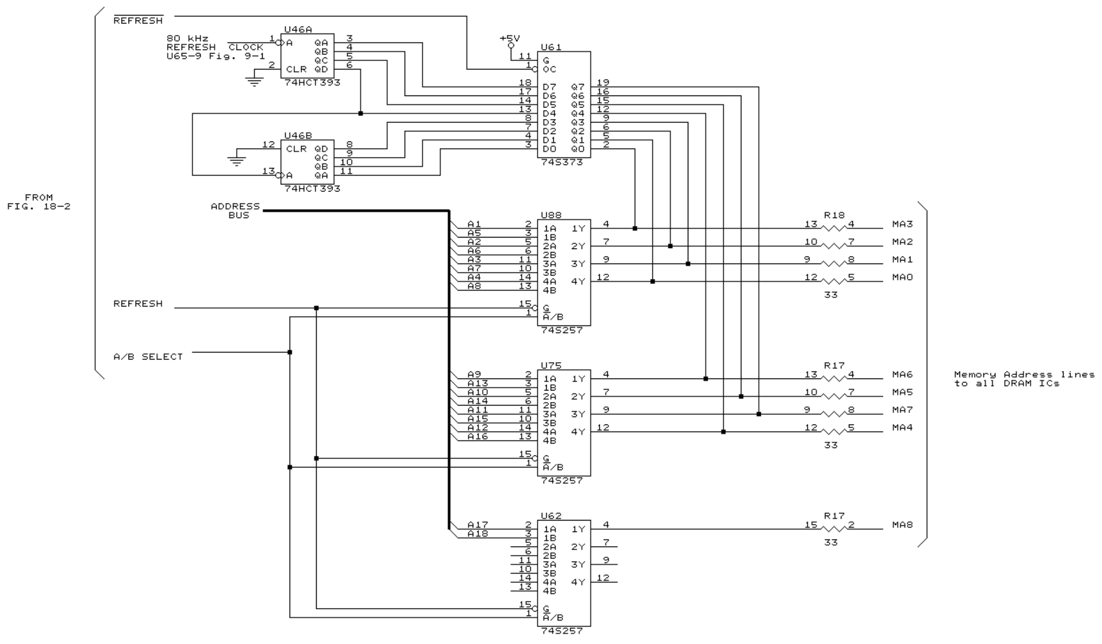
U52 is called a *150-nanosecond delay line*, and it does exactly what the name implies - it delays signals. It has several outputs, of which we only use pins 12 and 8. Whenever a logic signal is applied to its input on pin 1, it is delayed and appears on the outputs a specified delay time later. The total delay line is specified as 150 nanoseconds (ns), which means that the input comes out the last output, pin 8, 150 ns after it entered. But the delay line also has an intermediate output on pin 12, which provides only a 30 ns delay. (When you really think about it, 30 ns is a very short time. For example, a beam of light - the fastest thing we know of - only travels about 30 feet in 30 ns!)

Up until now, however, the input to the delay line was a low, and so both of its outputs have been low. U52 pin 8 therefore has been sending a low back to U51c pin 10 and U51d pin 13, which were needed to get everything started. U52 pin 12, on the other hand, has been sending out a low to flip-flop U49a, to U35B pin 4, and to the A/B SELECT line. After the 30 ns delay, however, U52 pin 12 goes high. This changes the A/B SELECT signal from a low to a high, and also clocks flip-flop U49a. Since pin 2 of the flip-flop is already high (since flip-flop U49B was reset and therefore REFRESH was high), the flip-flop now sets and sends out a low on $\overline{\text{CAS}}$ and DRAM DTACK.

Let's now jump ahead to Fig. 18-3 and see what is happening in the DRAM address multiplexers. As we mentioned last time, the DRAM address pins receive three separate addresses. During normal memory operation, they receive first a row address, followed by a column address; during refreshing, they receive a refresh address. The multiplexers shown in Fig. 18-3 select which of these three addresses is applied when.

Up until now, REFRESH has been low and $\overline{\text{REFRESH}}$ has been high. The circuitry at the top of Fig. 18-3 generates the refresh address, which can be sent to the DRAMs through a set of tri-state buffers in U61 if their $\overline{\text{OC}}$ (output control) input goes low. But since $\overline{\text{REFRESH}}$ is high, this keeps $\overline{\text{OC}}$ high and therefore prevents the refresh address from getting to the DRAMs.

Instead, the low REFRESH signal is applied to the $\overline{\text{G}}$ (gate) chip select inputs of U88, U75, and U62, enabling their circuitry. Each of these three ICs is a "quad two-input multiplexer", meaning that it contains four multiplexers, each having two inputs. The A/B select input on each IC selects


DRAM address multiplexers.

which of the two inputs is sent to the output of each multiplexer. Scanning down the inputs of U88, for example, A1 goes to output MA3 if the A/B select input is low, or A5 goes to MA3 if the A/B select input is high; similarly either A2 or A6 goes to MA2, depending on the A/B select input, and so on. Notice that the A inputs and MA outputs seem to be mixed up in a crazy order, which seems as though the memory is going to be very confused. In actual operation, it simply means that every time the 68000 stores something into memory it will go into what looks like the wrong location; but the next time the 68000 wants to read it back, it will be read back from the same wrong location and so the correct data will come back out.

At the beginning of the operation, A/B SELECT was low, and so each of the multiplexers chose one set of nine bits to send to the MA outputs. But after the delay line outputs a high A/B SELECT, the multiplexers switch and send the other nine address bits to the MA outputs. The first set of nine bits was the row address; the second set is the column address. Although it looks as though the column address comes out just 30 ns after the row address, actually the delay is somewhat greater. The row address is applied to the DRAM ICs as soon as the 68000 starts its memory operation; the column address is not applied until after the address decoder has recognized the DRAM address and sent out the $\overline{\text{RAM}}$ signal, which must then go through U51c and U37c before even entering the delay line.

Up until now, the operation has been as follows:

1. The 68000 sends out an address; since REFRESH and A/B SELECT are both low, nine bits of the address go to the DRAMs as a row address.

2. The address decoder sends out $\overline{\text{RAM}}$ which starts the DRAM circuitry.
3. $\overline{\text{RAS}}$ goes low.
4. After a short delay, A/B SELECT goes high and sends the other nine address bits to the DRAMs; this is now the column address.
5. $\overline{\text{CAS}}$ goes low.
6. DRAM $\overline{\text{DTACK}}$ goes low.

Note that $\overline{\text{DTACK}}$ normally signals the 68000 that a data transfer is completed, and yet the DRAM access is nowhere near being finished. In an effort to keep the computer going at maximum speed, $\overline{\text{DTACK}}$ is sent to the DTACK circuitry (U36, shown in Fig. 11-1) before the DRAM actually finishes; the 68000 doesn't respond for another few clock pulses and so the DRAM will have enough time to finish before the 68000 continues. Note that the timing of the entire DRAM circuitry is very carefully thought out; since most SK68K operations involve the DRAM, squeezing extra speed out of this circuit is very important.

Let's now return to Fig 18-2. After the 150 ns delay, U52 pin 8 goes high, which makes U51c pin 10 and U51d pin 13 both high. As a result, all inputs to U37c go high and so its output goes low. After passing through U66d, this makes $\overline{\text{RAS}}$ go high again. Meanwhile delay line U52 is now processing the low. After 30 ns, the low on U52 pin 12 makes A/B SELECT return low; it also changes the clock signal to U49a from high to low, but the flip-flop does not react since the 74ALS74 flip-flop only responds to a rising edge of the clock. Thus the flip-flop stays set, and continues outputting $\overline{\text{CAS}}$ and DRAM $\overline{\text{DTACK}}$.

After the delay line completes the 150-ns delay, its pin 8 goes low and the DRAM circuits are ready for another memory access. Meanwhile, the 68000 finishes its memory access also; it then turns off the AS address strobe, with the result that U49A finally resets, and both $\overline{\text{CAS}}$ and $\overline{\text{DTACK}}$ go back high or off.

RAM Operation During Refreshing

Let's now look at how the DRAM circuits operate when refreshing. First, the 80 kHz REFRESH CLOCK (from the bus error circuit of Fig. 9-1) is sent to the refresh counters, U46a and U46b, in Fig. 18-3. As mentioned last time, refreshing a DRAM involves reading 128 rows once every 2 milliseconds. Although only 128 rows are needed, U46a and U46b are each divide-by-16 counters, so together they make a divide-by-256 counter. All eight of their outputs are sent to the DRAM address lines through U61, even though only seven outputs are needed. The eighth output does no harm, however, and is there for possible future expansion.

Since the period of the 80 kHz signal is 1/80000 second, or 12.5 microseconds, the refresh counters complete a total of 128 counts in 128 x 12.5, or 1600 microseconds. This is 1.6 milliseconds, well within the ratings of the DRAM chips which require refreshing at least once every 2 milliseconds.

Once every 12.5 microseconds, when pin 9 of U65b goes high, this signal clocks U50b in Fig. 18-2, which therefore sets (since its D input is connected to +5 volts) to signal the DRAM circuits that it is time to do a refresh. The Q output of U50b therefore goes high, so that the next rising edge of the

MPUCLK signal (which might be anywhere from 8 to 16 MHz, depending on CPU speed) sets U50a. Its Q output now also goes high, sending a high to the D input of U49b. The circuit now waits for two other events: for the RSHAS pulse to go low (which is inverted into a high by U66B) and for MPUCLK to go low again. When this occurs, U49b finally sets, and this event finally starts the refresh cycle. Note that it was not sufficient for the 80 kHz signal to go high - the circuit waited for the correct sequence of MPUCLK and RSHAS before actually starting the refresh. The reason is that a refresh can interfere with the 68000's use of the DRAM memory unless the refresh is timed just right. This circuitry is designed to delay a refresh until the end of an address strobe; the idea is to try to do a quick refresh just after the 68000 has finished accessing memory and so try to squeeze the refresh into an unused time period. (During normal operation, the RSHAS signal - which comes from U63 in Fig. 10-1 - is the same as the AS strobe, so timing is synchronized with the end of AS. But when the 68000 is waiting for a $\overline{\text{DTACK}}$ from a board plugged into one of the expansion connectors, U63 substitutes a steady low to allow the DRAM circuitry to refresh without waiting for a real AS strobe.)

In any case, refreshing begins when U49b finally sets. This switches the REFRESH signal from a low to a high, and switches $\overline{\text{REFRESH}}$ from a high to a low. As shown in Fig. 18-3, when REFRESH goes high it turns off the three multiplexers (U62, U75, and U88); when $\overline{\text{REFRESH}}$ goes low it enables U61, so that the DRAM ICs receive the refresh address instead of the normal column or row address from the address bus.

Back in Fig. 18-2, however, the $\overline{\text{REFRESH}}$ signal is also sent to pin 1 of U51a. This starts exactly the same memory cycle as was started by the $\overline{\text{RAM}}$ signal from the address decoder, so that the delay line receives first a high and then a low, just as in normal operation. But this time there are two differences: first, the $\overline{\text{REFRESH}}$ signal is now low, and so the 30-ns output on pin 12 of delay line U52 cannot set U49a. Thus no $\overline{\text{CAS}}$ or DRAM $\overline{\text{DTACK}}$ is generated during a refresh. This means that only a row address is sent to the DRAMs, no column address. Furthermore, $\overline{\text{DTACK}}$ is not needed since the 68000 is not involved in refreshing, and doesn't even want to know that refreshing is occurring. The second difference is that the 30 ns output of the delay line is also sent to pin 4 of U35b. Since pin 5 is already high (because REFRESH is high), U35b pin 6 goes low, thereby resetting flip-flops U50b, U50a, and U49b. This ends the refresh cycle.

Finally, we need to look at how the 32 DRAM ICs are actually connected; Fig. 18-4 shows the connections to just U38, one of the DRAMs. First, each 256K DRAM IC has nine address lines which connect to MA0 through MA8. As shown in Fig. 18-3, these nine lines each come through a 33-ohm resistor, a rather unusual practice in normal digital circuits but actually quite common in memories. These resistors are used primarily to reduce overshoots and undershoots of voltage (above +5 volts and below 0 volts) which would otherwise exist on these address lines. The problem is basically caused by the fact that the memory address lines each go to 32 ICs. The wiring for these lines is therefore quite long and complex and thus represents a fairly large capacitance to ground. The buffers driving these lines would normally feed rather large current surges into this wiring, which would result in overshoots and undershoots. 33-ohm resistors are also

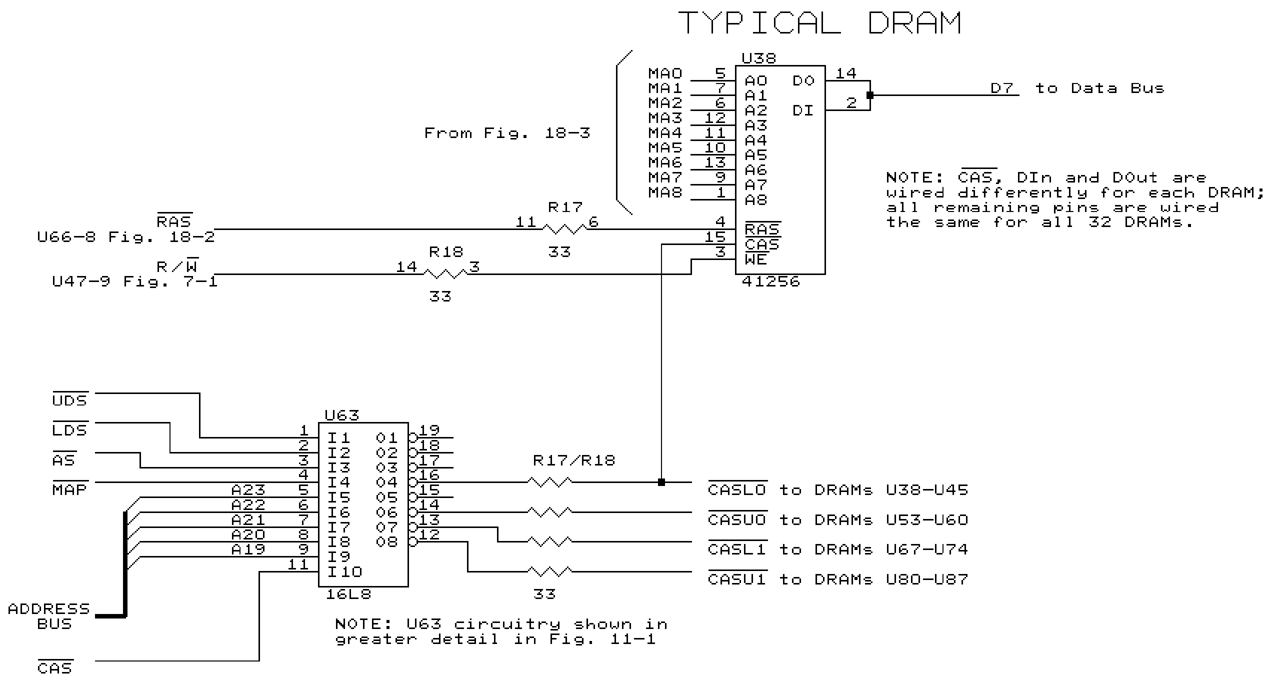


Fig. 18-4. DRAM IC wiring.

found on the R/$\overline{\text{W}}$ and $\overline{\text{RAS}}$ lines, which also go to all 32 DRAMs, as well as the $\overline{\text{CAS}}$ lines. Only the memory data lines do not contain the resistors, since each data line goes to only two ICs.

The $\overline{\text{RAS}}$ signal from Fig 18-2, and the R/$\overline{\text{W}}$ signal (from the 68000) are also applied to each and every DRAM IC in the entire system. That means that all DRAMs accept the same row address, and also all receive the read/write signal at the same time (although they don't actually read or write unless they also receive the $\overline{\text{CAS}}$ signal).

The $\overline{\text{CAS}}$ signal leaving Fig. 18-2 does not go directly to the DRAMs; instead it goes back to the address decoder (originally shown in Fig. 10-1, but also partly shown at the bottom of Fig. 18-4.) $\overline{\text{CAS}}$ is used only during actual memory accesses by the 68000, not during refreshing. At that time, the address decoder must decide which group of DRAMs is actually being addressed. The 32 DRAMs are divided into four groups of eight ICs:

(a) the even (or high-order) bytes of the first 512K, U53 through U60, which connect to $\overline{\text{CASU0}}$,
(b) the odd (or low-order) bytes of the first 512K, U38 through U45, which connect to $\overline{\text{CASL0}}$,
(c) the even (or high-order) bytes of the second 512K, U80 through U87, which connect to $\overline{\text{CASU1}}$, and
(d) the odd (or low-order) bytes of the second 512K, U67 through U74, which connect to $\overline{\text{CASL1}}$.

U63, the address decoder PAL, decides which group (or groups) of DRAMs to enable, based on the status of upper data strobe $\overline{\text{UDS}}$, lower data strobe $\overline{\text{LDS}}$, and address lines A19 through A23, and then passes the $\overline{\text{CAS}}$ signal to the appropriate group or groups. Note that it might send $\overline{\text{CAS}}$ to two groups of DRAMs at the same time if a 16-bit data transfer is needed from both an even and an odd byte at the same time.

Finally, within each group of eight DRAMs, each IC is connected to a different bit of the data bus so that every data transfer involves either eight or sixteen DRAMs, all writing or reading a different bit of the data bus. Since the DRAMs have separate data in and data out pins, these are connected together as shown in Fig. 18-4. Fig. 18-5 shows a pictorial view of the 32 DRAMs on the printed circuit board, identifying which IC connects to which data line, and which group connects to which $\overline{\text{CAS}}$ line.

18-2. Construction

It is now time to build the DRAM portion of the board. Begin by checking that the following components have been installed in prior steps:

U35, 74LS00 quad NAND gate and its socket # U37, 74LS10 triple 3-input NAND and its socket # U48, a 74LS08 quad AND gate and its socket # U51, a 74LS32 quad OR gate and its socket # U66, 74LS04 hex inverter and its socket # C68, a 33 pF disk capacitor # R12, a 10K resistor # R17 and R18, 33-ohm DIP resistor packs # U65, a 74LS390 dual decade counter and its socket.

Recheck C68 to make sure it is 33 pF, and not 0.1 µF like almost all the other disk capacitors on the board.

Now install the following:

U49	74S74 dual flip-flop and its socket
U50	74LS74 dual flip-flop and its socket
U52	150-ns delay line, soldered directly to the board
U62, U75, U88	74S257 quad 2-input multiplexers and their sockets. U62 uses a special IC socket with a built-in decoupling capacitor, as there was no room on the printed circuit board to place a separate capacitor right next to it.
U46	74HCT393 dual divide-by-16 counter and its socket
U61	74S373 8-input tri-state buffer and its socket. Make sure that this particular IC is not made by TI; if necessary, interchange with U19 to make sure it is made by a different manufacturer.

Although a TI-branded 74S393 works just fine as U19, the DRAM refresh circuitry has some very critical timing, and 74S373 ICs made by TI do not seem to operate well in this circuit - we have had good luck with units made by National Semiconductor and others.

We are now ready to install more components, but much of the wiring in this area is very close and we must take special precautions to guard against accidental short circuits. When installing the following components, install the IC sockets first, and after every group of eight or so, recheck the wiring and then turn on the computer to make sure it still works (make sure to remove all loose wires or solder before turning on the power). Although this is not a foolproof check, it does help to narrow down the cause of most problems as soon as possible after they happen. If at any point the computer suddenly stops working, recheck all new soldering and components and look for accidental solder joints. Here are the components to be installed next:

U38-U45 and U53-U60	Sixteen 41256 dynamic RAM ICs
U67-U74 and U80-U87	Another sixteen 41256 DRAMs if the optional second 512K of memory is to be installed.

Use 150-nanosecond ICs at 8 or 10 MHz, 120-nanosecond ICs at 12.5 or 16 MHz clock speeds. Sixteen ICs will provide 512K of RAM and is the minimum number that can be installed at this time. Even if you install only 16 DRAMs, it is a good idea to install all 32 sockets to avoid having to pull the board out of the cabinet later.

Next, install

C15-C46	thirty-two 0.1 µF disk capacitors near pin 1 of each of the 32 DRAM sockets
C47, and C49 through C54	seven 0.1 µF disk capacitors along the bottom edge of the board below the DRAM sockets
C56	0.1 µF disk capacitor between U75 and U88.

18-2a. Additional Construction Step

If you have an early production PC board, your circuit board has an error for which we apologize. As wired on the board, U48d-12 is connected directly to $\overline{\text{AS}}$, rather than going through U66b to RSHAS. This is a connection which worked well on the original system. But recent production XT-compatible monochrome video boards have a tendency to output a long wait signal to the bus while they complete internal operations. As originally wired, the SK68K would therefore delay $\overline{\text{DTACK}}$ while waiting for the video board to finish its operation. During this time, there would be no AS strobes, and therefore no refreshing.

As described earlier, the modified SK68K circuit now performs a refresh without waiting for AS while waiting for a video board. To implement this change (which may not be necessary except when used with new video boards), perform the following:

Cut the trace leading to U48d pin 12. Then install a jumper from U63 pin 15 to U66b pin 3, and another jumper from U66b pin 4 to U48d pin 12.

18-3. Testing

Next, let us try the memory out. Begin by moving the MAP jumper, J25, to position 1 to enable the DRAM, and then turn on the power. The speaker will probably beep and the normal HUMBUG prompt will probably appear on the screen, but this is not enough of a test - even if the memory malfunctions, it is still possible for all this to happen since HUMBUG uses mostly the static RAM; the DRAM is used only to hold some vector addresses.

To make sure the memory works we need to do some more tests. First, use the MT (memory test) command of HUMBUG to run a quick memory check. If you have installed all 32 DRAMs, then the correct command is

```
MT FROM ADDRESS 0  TO FFFFF
```

which checks out 1 megabyte of locations from address $0000 to $FFFFF. If you have installed only 16 DRAMs, then the correct command is

```
MT FROM ADDRESS 0 TO 7FFFF
```

which checks out only the first 512K of memory. In each case, press the space bar after the zero and after the last F to tell HUMBUG that you have finished typing the address. Either way, HUMBUG should print out a + sign and then its normal * prompt if the memory test passes.

Although the MT memory test of HUMBUG is fast, it is not extremely thorough. It merely goes through every location of memory and tries to set and reset every bit, one at a time. It then reads the bit back and tests it. But since it reads the bit back immediately after it sets it, the DRAM may seem to work even if the refresh circuitry is bad. We therefore need a more thorough test to check for a more long-term memory.

There are two ways to do this. One method involves copying the HUMBUG ROM into DRAM, waiting a minute or so, and then doing a memory compare to check the two against each other. Since the HUMBUG ROM is less than 20K in size, it is not big enough to fill up all of DRAM for a thorough test. Nevertheless, we can do a rough check by typing in the following two commands:

```
MO  ENTER OLD ADDRESSES: FROM F80000 TO F84000
    ENTER NEW ADDRESS: 1000
MC  REGION 1: FROM F80000 TO F84000
    REGION 2: 1000
```

The MO command moves the contents of ROM locations $F80000 through $F84000 down into RAM locations $1000 and up (do not store anything into locations 0 through $400, as these are used to hold other vector addresses.). The MC command, which should be done a minute or two later, does a memory comparison of the same two areas of memory. If the two sets of data do not match, then HUMBUG will display the addresses where a difference was found.

Some other possible tests are to use the FM (fill memory) command to fill all of memory with zeroes, and then the CS (checksum) command to check that the sum of all these zeroes is 00000000. Another way is to move whatever random data is in the bottom 512K into the top 512K, and then make sure that the checksum of the bottom 512K is the same as the top 512K. If you have plenty of time, you could also use the ROM-based Basic program to POKE consecutive numbers into memory locations, and then come back later, read them back with PEEK, and check against what was stored. There are many possibilities.

If all seems to work, then skip ahead to the next chapter; once we get to boot the SK*DOS disk operating system, there will be plenty of opportunity to test out the memory more thoroughly.

18-4. In Case Of Difficulty

Defective DRAM circuitry can show itself in many ways - the computer can be totally dead, or it might come to life but be unreliable, or it might appear to work but simply fails memory tests. The specific troubleshooting procedures vary, depending on symptoms. The most likely symptoms are:

1. The computer is completely dead. Move the J25 jumper back to position 2 to disable to DRAM and switch back to fully static RAM operation. If the computer is still dead, you have probably introduced a short circuit while soldering some of the memory sockets.

2. The speaker beeps, but nothing appears on the screen, or only the "please press enter" message appears on the monitor (unless you are using only a serial terminal), and at some point the HALT LED goes on. These symptoms generally indicate that at least part of the DRAM circuitry is working; the first step is to interchange the DRAM ICs, swapping each IC in an upper or even group with those in a lower or odd group. If the symptoms change then this generally indicates a defective DRAM IC. If, on the other hand, the symptoms stay the same, then the problem is probably elsewhere.

If an oscilloscope is available, then place J25 into position 2 to disable CPU accesses to DRAM, but keep the refreshing circuits going. Now check for the following, and trace signals if any of these appear wrong:

(a) An 80 kHz signal at U46 pin 1
(b) The frequency of each successive output from U46 should be half of the preceding output. For example, pin 6 should be at 40 kHz, pin 5 at 20 kHz, etc. (see Fig. 18-3).
(c) There should be thin, barely-visible (depending on the oscilloscope) negative-going pulses at U49 pin 8 and at U66 pin 8.
(d) There should be thin, barely visible positive-going pulses at U52 pins 8 and 12.
(e) U49a should stay set at all times so pin 6 should always be high.

3. If the computer works, but fails a memory test, try to analyze the MT test printout to determine where in memory the problem is. For example, if errors occur in locations $80000 and above, then the problem is only in the upper 512K of memory. If the problem occurs in a small region of memory, then the problem is likely to be only in a single IC.

Another way to narrow down defective DRAM problems is to use the ME command to store a number into memory at one of the locations flagged as defective by the MT test, and then read it back to see whether it was stored properly. For example, suppose the MT test indicates errors in odd locations between 4001 and 4FFF. This would indicate that an error is occurring in the odd or lower memory group in the first 512K of memory. Use the ME command to store the number $FF in location 4001, one of the

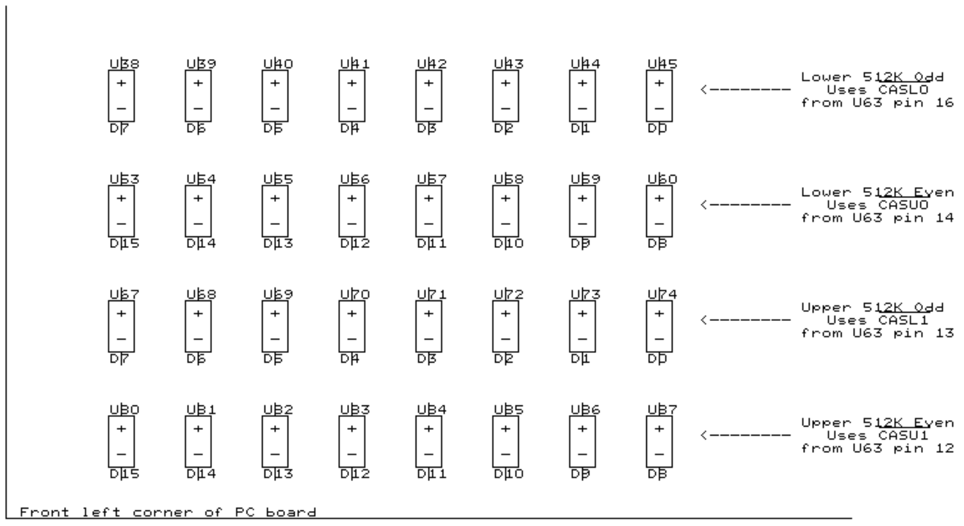


Fig. 18-5. Physical layout of the 32 RAM ICs.

defective locations. If you then read back $FB, for example, compare the bit patterns for FF (11111111) and FB (11111011). Since there is a difference in the third bit from the right (bit D2), you can then use Fig. 18-5 to identify the correct IC - it is U43, since this IC is connected to D2 in the lower 512K of memory. (Note that the bits in even locations are numbered from D15 on the left to D8 on the right, whereas the bits in odd locations are numbered from D7 on the left to D0 on the right.)

Chapter 19

Floppy Disk Controller

Without some "mass storage" device such as a floppy disk, even the largest computer would still be just a toy. The SK68K can use either a floppy disk or a hard disk; the floppy disk interface is right on the main SK68K board.

19-1. Discussion

Although we tend to look in awe at the floppy disk circuitry of any computer, actually it is a very simple circuit - because the most complex parts are hidden in a dedicated IC known as the FDC or Floppy Disk Controller.

Fig. 19-1 shows the circuitry. The heart of the circuit is the WD1772 FDC which handles most of the real work of the interface. On the CPU side, the FDC connects to the lower eight bits of the data bus, to address bits A1 and A2, to the R/$\overline{W}$ line, to the $\overline{RESET}$ line, to the $\overline{I/O4}$ select line from the address decoder, and to the 8 MHz clock signal CLK8.

The $\overline{I/O4}$ line comes from U34 in the address decoder (Fig. 10-1). It goes low whenever the 68000 accesses any address in the range from $FE0100 through FE013F, and thus selects the FDC whenever the 68000 does any read or write to any address in this range. But since only the lower eight bits of the data bus go to the FDC, only odd addresses in this range can actually be used for data transfer.

Internally, the CPU-side of the FDC is organized into six *registers* - six groups of flip-flops which store a byte-sized number. Two of these registers can be both written into and read by the 68000; another two can be read but not written, and the last two can be written into but not read. The 68000 selects which register it is reading or writing by (a) making the R/$\overline{W}$ line high for reading or low for writing, and (b) putting the appropriate bit pattern on address lines A1 and A2. These two bits are controlled by

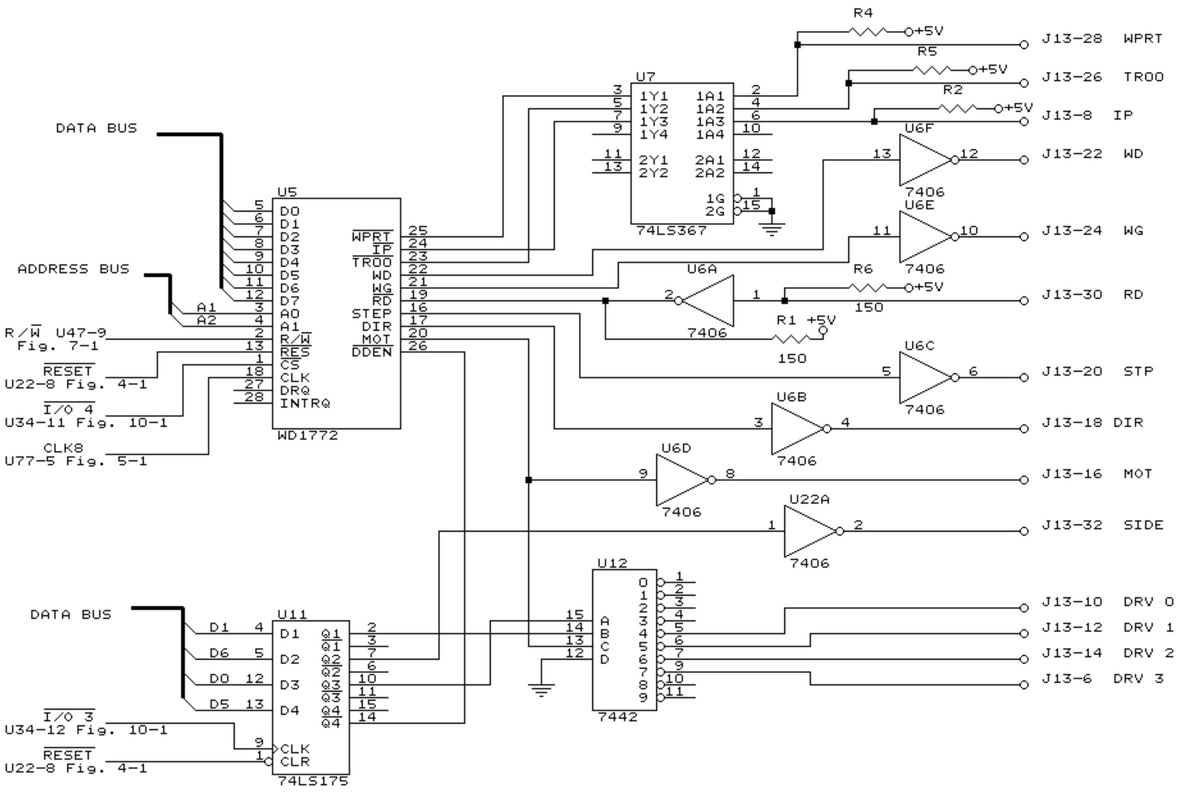


Fig. 19-1. Floppy disk controller.

choosing which address (in the range from $FE0100 through FE013F) the program accesses. For example, writing to location $FE0103 makes R/$\overline{W}$ low and puts the bits 01 on A2 and A1 respectively. (Remember that the 3 at the end of $FE0103 is the bit pattern 0011; the middle two bits of this pattern are A2 and A1, respectively.)

These six registers therefore appear to the 68000 at four address locations as follows:

$FE0101	Control register	Write only
$FE0101	Status register	Read only
$FE0103	Track register	Read / Write
$FE0105	Sector register	Read / Write
$FE0107	Write data register	Write only
$FE0107	Read data register	Read only

Each of the registers has a specific job:

To tell the FDC what to do, the 68000 (or, to be more exact, its program) puts a command into the Control register. The command might tell it to move to a specific track, to read or write a specific sector, or to format a track. (Read Appendix D about disk organization if you are not familiar with how data is stored on a floppy disk.)

To tell the 68000 (or its program) what is going on, the FDC puts status information into the Status register. Each bit in the status register has a function, such as to indicate that the FDC is busy, that it is waiting for data, that the disk is write-protected, or that an error has occurred.

The 68000 tells the FDC where to read or write by placing the track and sector number into the Track and Sector registers.

Finally, the 68000 places data to be written on the disk into the Write data register, or reads data from the Read data register.

Although the FDC IC handles most of the housekeeping involved with reading and writing floppy disks, there are two jobs it does not handle - choosing one out of several drives, or choosing a specific side of a double-sided disk. In addition, the FDC needs an additional input to tell it whether to use single or double density on the disk. All three of these jobs are handled by U11, a quad latch.

As shown in Fig. 19-1, the four data inputs of U11 connect to bits 0, 1, 5, and 6 of the data bus, while its CLK or clock input connects to I/O3. Like I/O4, I/O3 comes from U34 in the address decoder, but this signal is pulsed whenever the 68000 reads or writes to any address in the range of $FE00C0 through FE00FF. More specifically, any time that the 68000 stores a byte into location $FE00C1, U11 is clocked and any data that is on bits 0, 1, 5, or 6 of the data bus is stored into U11. The outputs of U11 are then used as follows: bits 0 and 1 go to U12, a decoder which selects one of four drives; bit 5 is sent to the $\overline{\text{DDEN}}$ pin of the FDC IC to choose either single or double density; bit 6 goes through U22a to the disk drives to select the side.

Finally, let's look at the connections to the disk drives themselves. Disk drives connect to the SK68K computer through a 34-wire flat cable which plugs into J13. Although not shown in Fig. 19-1, all of the odd pins of that connector are grounded, while the even pins carry signals to and from the drives. (On the connector, all the odd pins are on one side of the connector, while in the flat cable the odd-numbered wires alternate with the even-numbered wires. This means that between any two even-numbered signal wires there is always a ground wire which provides some shielding and isolation between signal pins.)

When there is more than one floppy drive, all the drives share the same 34-conductor cable and are connected in parallel. The only signals which are not shared by all drives are the DRV 0 through DRV 3 wires, which are used to select a specific drive. For example, to access drive 0, the SK68K places a low on the DRV 0 line while keeping the other three DRV lines high. When a command is then sent to the four drives, only the one selected drive actually obeys the command.

The signal wires can be grouped into three groups: read or write data, control signals to the drives, and status signals returned by the drives.

The two data lines are RD for Read Data, and WD for Write Data. Although data on the data bus travels in parallel, the FDC IC contains internal shift registers which convert it to and from serial data, which is then sent one bit at a time to or from the drive. Clock pulses are also added by the FDC to make sure that the data on the disk is properly timed and can be correctly read back. In single density, a clock bit is inserted between every data bit; in double density, clock bits are only inserted between consecutive zeroes. By cutting down on the clock bits, double-density operation can store more data bits on each track of the disk.

There are a number of control signals which are sent to the drive to control its operation. We have already discussed the four DRV signals, which select a particular drive, and the SIDE signal which selects the side of the disk. Other control signals include WG, the Write Gate signal, which goes low to tell the drive to switch from reading to writing; STP (step) which tells it to move the head from one track to the next; DIR which tells it which direction to move the head in; and MOT, which tells it to turn the motor on.

By way of explanation, MOT actually controls all drives, so even drives not selected (with a DRV signal) turn their motors on and rotate the disk. The STP signal is so named because floppy drives use a *stepper* motor to move the head back and forth between tracks. A stepper motor has a ratchet-like motion which is so aligned that the head settles over the correct track when the motor clicks into the next position. In this way the precise positioning of the head over a track is handled by the motor without any external help.

Finally, each drive sends back three status signals, all usually derived from photo-electric sensors in the drive. A drive sends back a low on the WPRT (write protect) line if it senses that the write-protect notch on the disk is covered with tape. It sends back a low on the TR00 (track 00) line when it senses that the head has been stepped all the way to track 0, the outermost track on the disk. (Note that the FDC has no easy way of knowing which track the head is positioned over - it keeps track of the head position by sensing track 0, and then keeping a count of the STP step pulses as the head moves in and out.) The IP or Index Pulse signal carries a low pulse every time that a disk rotates so that the beginning of the track is under its head. The drive senses the beginning of the track by sending a beam of light through a small hole in the disk called the *index hole*; once every revolution of the disk, the light is sensed and generates the IP pulse.

The FDC uses the index pulse in several ways. In normal operation, the presence of the pulse signal tells the FDC that there is a disk in the drive and the door is closed (otherwise the disk would not turn). While formatting the disk, it tells the FDC where to begin and end each track. When the FDC encounters a disk error, it also uses the IP signal to count down how many times it retries the operation before it gives up.

19-2. Construction

Install the following components:

U5	WD1772 floppy disk controller, and its socket
U11	74LS175 quad latch, and its socket
U12	7442 BCD-to-decimal decoder, and its socket
U7	74LS367 hex bus driver, and its socket
U6	7406 open-collector hex inverter, and its socket

R1, R2, R4, R5, and R6	150-ohm 1/4-watt resistors
J13	a 34-pin header
C6	0.1 µF disk ceramic capacitor

19-3. Testing

Most 3-1/2" or 5-1/4" floppy disk drives will work with the SK68K, but a double-sided drive (and, if possible, an 80-track drive) is preferable as it will hold more data. In the PC world, these drives are normally identified as 360K for 40-track drives, or 720K for 80-track drives; drives labelled 1.2 meg or 1.44 meg are not suitable.

Note also that the SK*DOS disk operating system is normally supplied on 80-track double-sided disks unless you specify otherwise. An 80-track drive can read 40-track disks by a process called double-stepping, where the drive takes two steps of 1/96" to move the 1/48" spacing between tracks on a normal 40-track disk; SK*DOS automatically tries double-stepping when it encounters a disk error, so reading a 40-track disk in an 80-track drive is totally invisible to the user. But the reverse is not true - although SK*DOS can write a 40-track disk in an 80-track drive, it often happens that this makes the disk unreadable on a 40-track drive. The reason is that the tracks on an 80-track disk are not just closer together, they are also narrower. When an 80-track drive writes on a 40-track disk, it does not write over the full track width and some of the original 40-track data may still remain around the edges of the track. When a 40-track drive subsequently reads the disk, it reads the new data as well as some of the old. Depending on the exact track positioning and other factors, it may then misread the disk and fail.

The next task is to make sure that the jumpers on the drive are properly set. As described above, selecting one out of a possible four drives is done by the four DRV lines. Although all four lines go to each drive, only one DRV signal is actually used by any single drive. Fig. 19-2 shows a simplified diagram of some of the jumpers which control drive selection on a typical floppy drive. As you can see, the four DRV signals appear on pins 6, 10, 12, and 14 of the 34-pin connector, and go to a set of jumpers usually called DS0

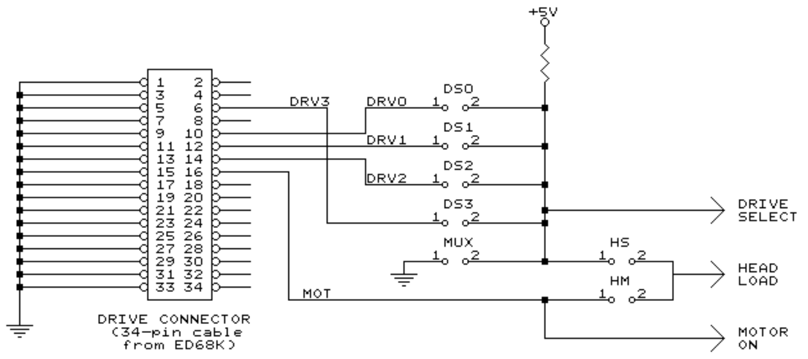


Fig. 19-2. Drive select circuitry in a typical drive.

through DS3 (although sometimes they are labelled DS1 through DS4). When one of these jumpers is installed, then this DRV signal goes to the 'drive select' line on the drive to enable the drive.

To use a single drive with the SK68K, you must make sure that it has only the DS0 jumper installed. In some drives, the connections are made through a shorting plug installed in an IC socket; in this case you may have to break three of the four connections on the shorting plug (or install a small DIP switch instead of the shorting plug). In other cases, you may have to move a jumper from one of the other positions into DS0. (Many floppy drives currently sold for use in IBM PCs or clones have the DS1 jumper installed, since the drive selection in these systems is done by flipping wires in the 34-pin cable, rather than by moving jumpers on the drive.) If you install more than one drive, place each DS jumper in a different position, starting with DS0 for your main drive.

In addition to moving the DS jumpers, you should also remove the MUX (or MX) and HM jumpers if installed, and place a jumper in the HS position. MUX is used to permanently select a drive for those computers which do not provide DRV signals, while the HM and HS jumpers control the 'head load' signal in those drives that have a solenoid which moves the read/write head against the disk. Installing the HM (head with motor) jumper would bring the head against the disk as soon as the motor turns on, whereas installing the HS (head with select) jumper only brings the head against the disk if the drive is selected. (Many modern drives do not have a head load solenoid, in which case this item does not apply.)

One last item on the drive(s) concerns small resistor packs installed in sockets. In order to minimize noise, each of the wires bringing a signal from the computer to the disk drive must be properly terminated with a resistor. Drives usually have small resistor packs which provide the terminations. But to avoid overloading the signals when more than one drive is installed, only the one drive at the very end of the cable should have its resistor pack installed. If there is no terminating resistor pack, or if there are two or more, the disk system may be unreliable, so check each drive in your system. (If you are using used drives, they may be missing the resistor packs altogether.)

Once this is completed, connect the four-pin power plug from the power supply to the drive, and connect the 34-pin cable between the computer and disk drive. Pin 1 of J13 is toward the back of the board, and the pin 1 end of the connector is marked on most disk drives with a small notch between pins.

Finally, insert a blank disk into whichever is drive 0 and turn on the power. Once the HUMBUG program is running, examine the disk drive to check that the motor is off and the drive select LED on the face of the drive is off. If either of these is on, the 34-pin cable is most likely connected backward at one end.

Now type the HUMBUG command FD (floppy disk). Since the drive still contains a blank disk it cannot boot SK*DOS, but still the drive motor should start and the drive select LED should light. If the motor comes on but the LED does not, the DS jumpers may not be properly placed. Don't go on until the motor and LED behave properly.

Once you have made all these checks, reset the computer, remove the blank disk, place a write-protect tape strip on the write-protect notch of your SK*DOS system disk, and place it in the drive. Then type the FD command.

If all goes well, the disk drive should start, the drive select LED should go on, the head should go back and forth a few times, and in a few seconds the SK*DOS signon message should appear. Congratulations - your system is working and almost finished!

19-4. In Case Of Difficulty

If not, then a bit of debugging is in order. First check the type of disk drive - a single-sided drive cannot read a double-sided disk, a 40-track drive cannot read an 80-track disk. If you are really quick, try to count the disk revolutions per second (rps) - the disk should be turning at 5 rps, not 6. If it turns at 6 rps, it may be a 360 rpm 1.2 megabyte drive intended for an AT-type system; if so, all is still not lost since some of these drives have a jumper to select either 300 or 360 rpm.

If the drive seems correct, use HUMBUG's FM command to fill all of memory from address $0800 through $8000 with zeroes, and then try the FD command again. After about 15 seconds, reset the system and use the HA command to look at locations $0800 through 0900. If these locations still contain zeroes, then the disk system totally failed to read the disk. If an oscilloscope or logic probe is available, check that the IP and RD lines are normally high, but have negative-going pulses just after you type the FD command. If not, then the disk may be in the drive backward, the disk drive may still not be properly selected, or may be defective. An easy way to check is to remove the disk, turn off the power, manually (and very carefully - it might be useful to ask a knowledgeable person for help!) move the drive's head carriage a half-inch toward the center of the disk, and then try again. As soon as you type the FD command, the FDC should step the head carriage outward toward track 0; if not, then some of the control signals are either not getting to the drive, or else are being ignored.

If locations $800 and up are now nonzero, then *something* was read from the disk. Check whether the first few bytes at $0800 are 60 08 50 54 32; these are the very first bytes read from track 0 sector 1 of the disk. If this data is there, then the disk system is almost OK. The data read from this very first sector contains the first half of a program we call the *superboot*, and loads into locations $0800 through $08FF. Once loaded, HUMBUG's FD command then jumps to location $0800, and this program loads the very next sector (track 0 sector 2) into memory at location $0900, so check whether the area from $0900 through $09FF is now filled, or whether it still contains zeroes.

The combination of these two sectors is now supposed to load the SK*DOS.SYS system file into memory and jump to its beginning at location $1000. Look at locations $0805 and $0806; these two bytes should contain two nonzero numbers, which tell the program where on the disk to find the SK*DOS.SYS file.

Next, look at locations $1000 through 1100. If these locations still contain zeroes, then the SK*DOS file was not read. If the drive seems able to read the superboot program but fails to read SK*DOS.SYS, there may be a problem with the STP or DIR lines or circuitry so that the drive can read track 0 but not other tracks. This symptom would also appear if you were using a 40-track drive to read an 80-track disk, or a single-sided drive to read a double-sided disk.

Finally, look at memory locations in the range from $4000 to $5000 or so. If these are still zeroes, then an error may have occurred while reading the disk. A common problem is head alignment - the fact that the disk head may not be centered over a track. Try a different drive; another possibility is to copy your SK*DOS system disk on another system and try the copy. For example, SK*DOS disks can be duplicated using the Copy II PC Option Board on PC / XT clones. Alternatively, exchange your SK*DOS disk for a new one.

Chapter 20

Parallel Printer Port

There are several ways to connect a printer to the SK68K - a serial printer can be connected to the serial port at J21, a parallel printer can be connected to the printer port on a monochrome video card, or a parallel printer can be connected to J8 on the SK68K system board. All of these are supported by SK*DOS drivers, and it is even possible to connect all three printers at once. This step discusses the parallel printer port at J8.

20-1. Discussion

As Fig. 20-1 shows, there isn't actually much to discuss - the entire port consists of just U2, an MC68230 parallel interface / timer IC (PI / T) which is part of the Motorola 68000 family. The 68230 consists of essentially three

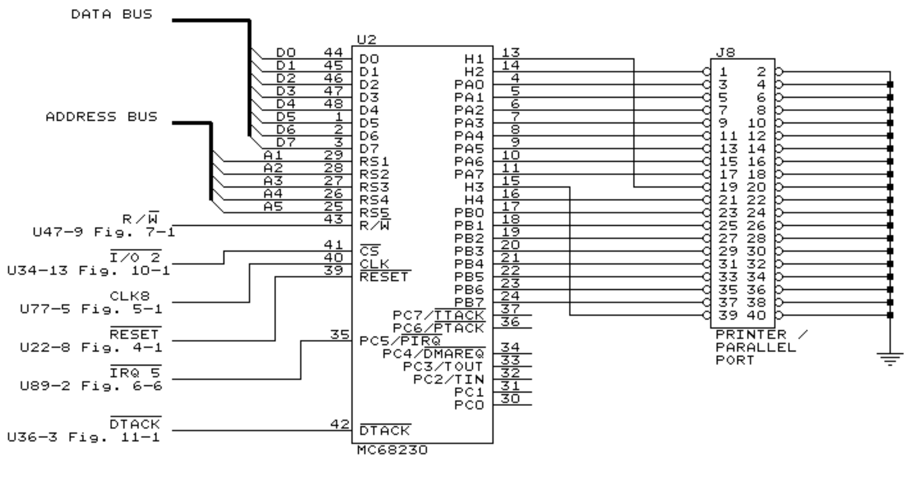


Fig. 20-1. Printer / parallel port.

parts:

Port A is an eight-bit input/output port which uses the eight PA pins along with two *hand-shaking* pins called H1 and H2. The PA pins are data pins for the eight-bit parallel data, and can be used for both input and output; H1 and H2 are used for hand-shaking; that is, for sending control information. When used with a printer, port A (via pins 1-20 on J8 - pins 21-40 are not used in this case) connects to a Centronics-compatible parallel printer connector. The PA lines carry characters to the printer, H2 tells the printer that a character is ready, and H1 lets the printer tell the SK68K whether it is ready or busy.

Port B is a second eight-bit port similar to port A but using PB, H3, and H4 lines instead of PA, H1, and H2. Although it could also be used to drive a printer, in most cases it will not be connected (when using port A to drive a parallel printer, don't connect to J8 pins 21-40.)

The third part of the 68230 PI/T is a timer. This is a 24-bit counter which can count either CLK8 clock pulses or external pulses, generate square waves, generate interrupt signals to the computer at fixed time intervals, and various other functions. It is not used by SK68K software, but there is no reason why you can not use it in your own programs.

On the CPU side, the 68230 connections are similar to those of the floppy disk controller. It receives the $\overline{I/O2}$ signal from the address decoder, so it is selected whenever any address in the range of $FE0080 through $FE00BF appears on the address bus. Like the FDC, it contains a number of internal registers which are accessed at specific addresses, depending on the status of address lines A1 through A5. But there are just too many registers and operating modes to describe here - Motorola publishes a 75-page manual on this IC alone!

20-2. Construction

Install the following components:

U2	MC68230 PI/T and its socket
J8	40-pin header
C2	0.1 µF disk capacitor

Fig. 20-2 shows how to construct the printer cable with 20-conductor flat cable. You may use a 20-pin connector at the computer end if you make sure to install it only in the pin 1 end of J8; otherwise, use a 40-pin connector and make sure to attach the 20-wire cable all the way toward the pin 1 end. Place a standard Centronics printer connector at the other end, making sure that pin 1 of this connector connects to pin 1 of the other connector.

20-3. Testing

Connect a standard Centronics-compatible parallel printer to J8, and then boot SK*DOS (with the FD command).

To use the printer, you must first install the parallel printer driver. From the SK*DOS prompt, use the command

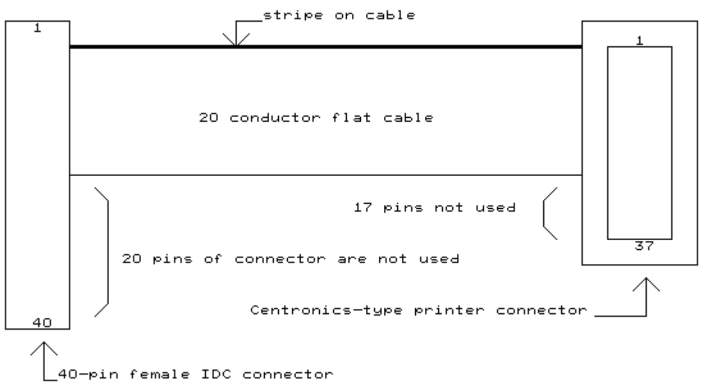

stripe on cable
1
1
20 conductor flat cable
17 pins not used
37
20 pins of connector are not used
Centronics-type printer connector
40
40-pin female IDC connector

Fig. 20-2. Parallel printer cable.

```
DEVICE PARALLEL AS PRTR AT 2
```

This installs the PARALLEL.DVR driver from the disk as device 2 and gives it the name PRTR. To try out the printer, try any SK*DOS command, but insert the text >PRTR on the command line to redirect the output from the screen to the printer. For example, the command

```
ACAT >PRTR
```

would print an alphabetized catalog of the disk on the printer.

Chapter 21

Optional -HDO Hard Disk Port

Rejoice - this is the last section of your SK68K computer to be wired. In fact, most users will probably choose to omit this section.

21-1. Discussion

The cheapest way to add a hard disk to your SK68K is by using a Western Digital WDXT-GEN (or -WX2) type hard disk controller, plugged into one of the XT-style connectors on the back. These controllers cost about $70, or can be obtained complete with a 20-megabyte disk and cables for under $250.

Still, there are some users who wish to use a more expensive controller, the Western Digital WD1002-HDO which alone costs about $250, because they want to keep all their XT-type slots open for other purposes. The -HDO hard disk port shown in Fig. 21-1 is for them; all others can simply skip this part since the -HDO controller has no other advantages. (While both hard disk controllers can be installed at the same time, they require slightly different versions of SK*DOS and therefore cannot be used at the same time.)

The -HDO controller connects to the SK68K with a 40-pin flat cable connected to J17. It needs the data bus, three bits from the address bus to select internal registers (buffered address lines BA0 through BA2, obtained from the XT-type interface connectors are used), the $\overline{\text{I/O5}}$ signal which selects this port at addresses $FE0140 through $FE017F, $\overline{\text{RESET}}$, and a pair of signals called $\overline{\text{WE}}$ and $\overline{\text{RE}}$ for write enable and read enable, which are generated by U15b and U15c from the R/$\overline{\text{W}}$ signal and a port-select signal derived from U16.

U16 is essentially a digital delay used to generate $\overline{\text{DTACK}}$. Each time the -HDO port is selected by $\overline{\text{I/O5}}$ and the AS address strobe signal arrives, U16 starts to shift the AS signal from stage to stage. After two CLK8 clock

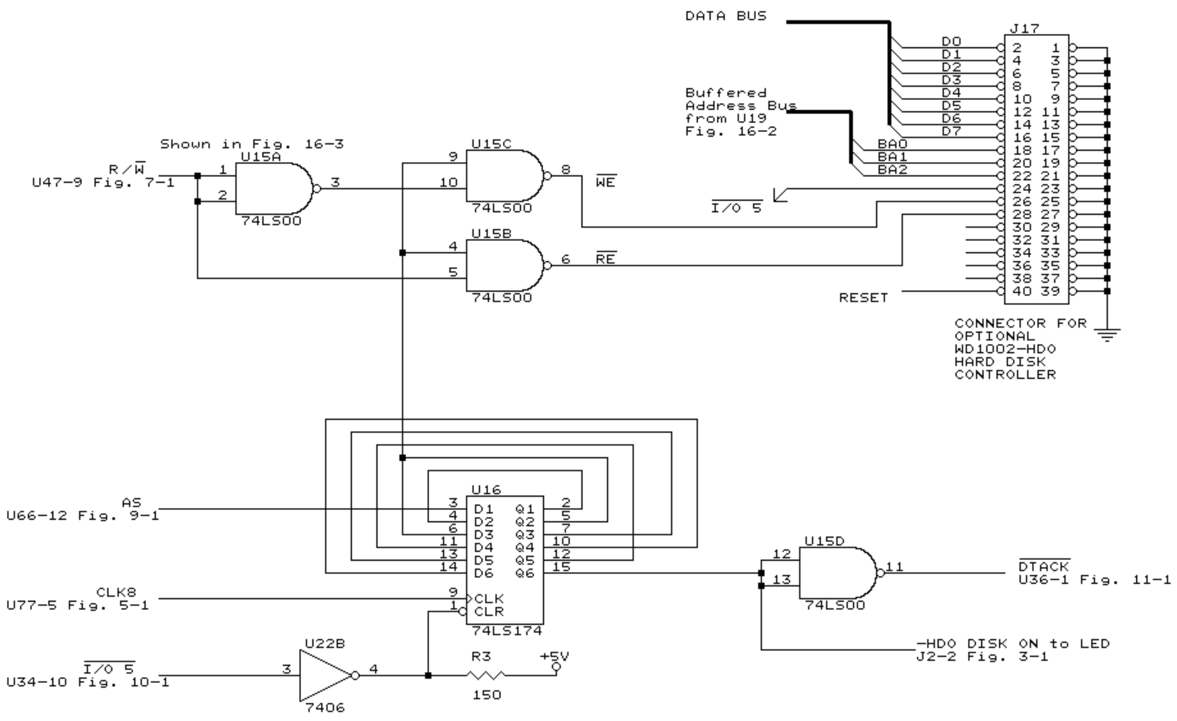


Fig. 21-1. -HDO hard disk controller port.

pulses, it sends a signal up to U15b and U16c to generate either $\overline{\text{WE}}$ or $\overline{\text{RE}}$, depending on whether the system is reading or writing. After six clock pulses, it sends a pulse to U15d, which then generates $\overline{\text{DTACK}}$.

21-2 Construction

First, **remove the jumper between U15 pin 7 and pin 12.** When we installed U15 in Chapter 16 as part of the XT-type bus connector circuitry, we put in this jumper to prevent U15d from generating a constant $\overline{\text{DTACK}}$ which would prevent the rest of the computer from working properly. Then install the following parts:

U16	74LS174 hex type D flip-flop and its socket
R13	150-ohm 1/4-watt resistor
J17	40-pin header

Other components, such as R10, U15 and U22 have been installed in previous steps, so this completes the installation.

21-3. Testing

There is no easy way of testing this port if you do not have an actual -HDO controller.

If you do, then plug in the controller with a 40-pin cable and start the computer. First, boot SK*DOS from a floppy disk using the FD command. If your hard disk is empty, you will have to format it with the HDFORMAT command; if it already has files on it, then use the DRIVE command to assign it a drive number and check its contents with the DIR command.

Chapter 22

Loose Ends

If you still have a test wire connected to pin 1 of J14, cut it very short and use it to jumper pin 1 to pin 2. Since we will no longer need our LED logic probe, this restores the circuitry feeding the LED.

Way back in Chapter 3, you installed three LEDs at J15, J16, and J17. If you install the SK68K in a cabinet which has some LEDs on the front panel, you may wish to replace the board-mounted LEDs with those on the front panel. Since the DSK LED at J16 is only used with the -HDO hard disk controller, you will probably choose to leave it, but the POWer LED at J15 and the HLT LED at J17 might be useful on the front panel. In that case, cut off these two LEDs, leaving about 1/2" of their leads sticking up above the board. The two-pin connectors that most cabinets are supplied with for LED connections slip over the stubs of the LED wires (if the LEDs do not light, reverse the leads.)

Look over the printed circuit board carefully to make sure that all solder connections are right and that there are no areas which will suddenly cause problems a few months from now. Check also to make sure that all components are installed; look especially for capacitors or resistors which you may have missed. If some of these are missing, the computer may still work but may not be as reliable as it should be.

Before mounting the printed circuit board in a cabinet, note again that the board mounting holes have copper lands both on the top and bottom of the board. These lands are not at the same potential! The lands on the bottom are grounded, but the lands on top connect to +5 volts. You must therefore use non-conductive mounting hardware to prevent shorting the +5-volt line to ground.

Appendix A

SK68K Parts List

	Integrated Circuits
U1	74LS245 Octal bus transceiver
U2	MC68230P8 Peripheral interface / timer
U3	3.68 MHz oscillator
U4, U10	MC68681 DUART
U5	WD1772 floppy disk controller
U6, U22, U32	7406 Hex inverter/buffer (o.c.)
U7	74LS367 Hex bus driver
U8, U29	1489 RS-232 receiver
U9, U30	1488 RS-232 driver
U11, U24, U31, U33, U76	74LS175 Quad D flip-flop
U12	7442 BCD decoder
U13, U50	74LS74 Dual D flip-flop
U14, U26, U51	74LS32 Quad 2-input OR
U15, U35	74LS00 Quad 2-input NAND
U16	74LS174 Hex D flip-flop
U17, U18	74LS373 Octal latch
U19, U61	74S373 Octal latch
U20, U27	27128 16Kx8 450ns EPROM (or 27256 or 27512)
U21	6116 2Kx8 400ns static RAM
U23, U49	74S74 Dual D flip-flop

U25	74LS322 8-bit shift register
U28	6116 2Kx8 400 ns Static RAM or MK48T02 clock
U34	74LS138 3-to-8 decoder
U36	74LS30 8-input NAND
U37	74LS10 Triple 3-input NAND
U38-U45, U53-U60, U67-U74, U80-U87	256K 150ns dynamic RAM **
U46	74HCT393 Dual 4-bit counter
U47	MC68000P8 microprocessor **
U48	74LS08 Quad 2-input AND
U52	DDU66-150 150ns Delay Line **
U62, U75, U88	74S257 Quad 2-input multiplexer
U63	16L8 programmable array logic
U64	74LS139 Dual 2-to-4 line decoder
U65	74LS390 Dual decade counter
U66	74LS04 hex inverter
U77	74ALS74 Dual D flip-flop
U78	16 MHz oscillator
U79	Optional 20 (or 24 or 32) MHz oscillator **
U89	74LS148 8-to-3 line priority encoder
U90	74LS164 8-bit shift register
U91	555 timer
U92	optional 14.313 MHz oscillator
	Connectors
J1-J6	62-pin card edge connector
J7, J8	40-pin dual header strip
J9	5 pin PC DIN connector
J10A, J10B	6-pin power connector
J11, J12, J21, J22	6-pin dual header strip
J13	34-pin dual header strip
J18	4-pin single header strip
J19, J20, J24, J25	3-pin single header strip
J23	2-pin single header strip
	Resistors (all 1/4 watt, 10% unless otherwise noted)
R1-R6	150 ohms
R7	4700 ohms

R8-R10, R12, R13	10,000 ohms
R14, R15, R16	330 ohms
R17, R18	33-ohm 16-pin DIP package
R19	10,000-ohm 8-pin SIP package
R20, R21, R24, R26	2200 ohms
R22, R23	1 megohm
R25	33 ohms
	Capacitors
C3, C4, C5	47 pF disc ceramic
C63	1 µF 16-volt tantalum
C90	10 µF 16-volt tantalum
C68	33 pF disk ceramic
all others (quantity 64)	0.1 µF disc ceramic
	Integrated circuit sockets
1	8-pin
22	14-pin
47	16-pin
1	16-pin with decoupling capacitor
7	20-pin
2	24-pin
3	28-pin
2	40-pin
1	48-pin
1	64-pin
	Miscellaneous components
1	SK68K printed circuit board
1	Cabinet (PC, XT, or AT clone)
1	135- or 150-watt Power supply (PC or XT clone)
3	LEDs
4	shorting plugs

NOTE: At higher CPU clock rates, components marked with ** must be replaced with faster versions.

Appendix B

Computer Number Systems

Internally, computers do all their calculations with binary numbers which may only contain the binary digits (bits) 0 and 1.

Imagine that you had to count pages in a book, but were told that you were not allowed to use any number containing the digits 2 through 9. In other words, your numbers would only be allowed to contain zeroes and ones. You would then count like this:

1
(skip 2 through 9 because they contain forbidden digits)
10
11
(skip 12 through 99)
100
101
(skip 102 through 109)
110
111
(skip 112 through 999)
1000
1001
(skip 1002 through 1009)
and so on.

Thus the first page would be numbered 1, the second would have the number 10, the third would be numbered 11, and so on. In other words, we would get a table like this:

Page	Binary Count
1	1
2	10

Page	Binary Count
3	11
4	100
5	101
6	110

You may think of the left column as being a normal decimal number, while the right column is its binary equivalent. When you extend the table to include zero and also some larger numbers, you get a table like the one below (we've added a few extra zeroes, but that doesn't make a difference. After all, 1, 01, 0001, or even 000000001 are still just 'one'.)

Decimal	Binary	Decimal	Binary
0	0000	9	1001
1	0001	10	1010
2	0010	11	1011
3	0011	12	1100
4	0100	13	1101
5	0101	14	1110
6	0110	15	1111
7	0111	16	10000
8	1000	17	10001

Suppose you were limited to numbers consisting of just four bits. Then the smallest number would be 0000 (decimal zero), while the largest number would be 1111, or the decimal number fifteen. But when you want to go to sixteen, you need one more bit. So we see that you need more and more bits as you count to higher and higher numbers. In fact, binary numbers are roughly three times as long as their decimal equivalents - the decimal number 999 translates to a binary 1111100111, while the decimal 9999 translates to 10011100001111.

Which brings us to the binary numbers that travel on the data bus and address bus of the SK68K computer. With a 24-line address bus, the smallest address in the SK68K computer consists of 24 zeroes, while the largest address consists of 24 ones. All the other addresses have some intermediate combination of zeroes and ones such as, for example, 110101011100001111101010. Although such an address is fairly easy for the computer to handle, we humans find it hard to read or write such long numbers without making a mistake. Hence we use a mathematical shorthand called hexadecimal notation instead.

To use hexadecimal numbers, we first split the binary number into groups of four bits like this:

```
1101 0101 1100 0011 1110 1010
```

Now we want to convert each group of four bits into a single digit, but right away run into a problem: the very first 1101 is equal to decimal 13, which is two digits, not one. In fact this is going to happen with 10, 11, 12, 14, and 15 as well. And so we extend the decimal number system with six additional digits (called *hexadecimal digits*) A through F. We write A instead of 10 (decimal ten), B instead of 11 (decimal eleven), and so on, up through F for 15 (decimal fifteen). The number is now written as

```
1101 0101 1100 0011 1110 1010
 13    5   12    3   14   10
  D         C         E    A
```

or just plain D5C3EA. That gives us this conversion table to use between binary and hexadecimal:

Hexadecimal	Binary	Hexadecimal	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1101
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Returning to the address bus ... the smallest address is 24 zeroes; splitting into groups of four bits we get six groups of 0000, which translates into the hexadecimal (also called just hex) 000000. The largest address is 24 ones, which separates into six groups of 1111, or a hex address of FFFFFF.

One last item - when you see a number such as 3152, it is sometimes hard to tell whether this is supposed to be a decimal number or hexadecimal. Whenever there is any doubt, users of Motorola processors always put a $ before hex numbers. Thus $3152 would be hexadecimal, whereas a plain 3152 would usually be decimal (except if someone forgot to write the dollar sign, or else if it is obvious from the context.)

Appendix C

How To Solder

Even if you already have experience soldering components to a printed circuit board, this Appendix contains useful information to help you avoid disaster in wiring your SK68K computer trainer.

Using a printed circuit board for a project eliminates most worries about connecting components to the wrong place, but it introduces new problems, especially if the board contains many components and very small wiring, as the SK68K computer board does. Bad construction technique can result in short circuits, bad connections, or even a ruined pc board. If you have never soldered on a printed circuit board before, find an expert technician or repairman and ask him or her to show you how it's done. Even your local TV repairman may be a good source of expertise. This Appendix describes some of the hints and tricks, but there is no substitute for on-the-job training. You just don't want to train on the SK68K computer pc board!

Good soldering requires (a) a steady hand and some patience, (b) good eyesight, (c) the right equipment, and (d) the right technique. Most of us can manage the steady hand, resting our hand on a thick book or the table edge if need be, and the patience we can't help you with. In the absence of good eyesight, we recommend one of the magnifiers which slips over the head. It is available at many stationery stores and also from Heathkit. It slips over the head and puts a magnifying lens in front of each eye. Highly recommended.

On to the equipment. First, you need the right solder. NEVER USE ACID CORE SOLDER because it corrodes the connections. The proper solder for electronic use is called rosin core. Solder is made of a combination of lead and tin; the best solder is 60% tin and 40% lead, although so-called 50-50 solder will also work. But beware - some solders are labelled 60-40 to make you think they are better, but are in fact worse than 50-50 because they contain more lead rather than more tin. The composition of solder is important because the correct 60- 40 solder melts at the lowest temperature and is thus easiest to use.

Next, you need the right soldering iron. A pencil type iron rated 35 watts or so is good; a *temperature controlled low-voltage soldering station*, made by Weller, Ungar, and others, is a more expensive but ideal choice. The cheaper non-temperature controlled irons sometimes tend to run too cool (which results in inadequate joints) or too hot (which may burn the board.)

Do not get anything over 45 watts, and definitely do not use a solder gun. It is almost certainly going to run too hot. Aside from possibly burning the board, a hot tip corrodes much faster and gets much dirtier in use. Speaking of tips, the slightly more expensive iron-clad tips last much longer than pure copper tips, and also stay cleaner.

Cleanliness is important. Use a wet sponge to clean off the iron tip every few minutes or so. We use a sponge in a holder on the solder stand, and wipe the iron on it just before using it. This avoids the problem that any dirt on the soldering iron's tip (which is usually corroded solder and rosin flux) will flow onto the soldered joint and contaminate it. Dirt also prevents good contact between the iron and the joint, which is needed to quickly melt the solder without overheating the joint.

Once wiped clean, the iron sometimes needs three or four seconds to heat up again, and also needs just a bit of solder to 'wet' the tip with solder and make it shiny.

Most soldering instructions claim that you should put the iron on one side of a joint, hold solder against the other side of the joint, and wait for it to melt and flow on the joint. We find this doesn't always work. When a clean iron tip touches the joint to be soldered, it often touches at just a single tiny point; the result is that not enough heat gets transferred to the joint to melt any solder. You can try to get around this by placing the flat part of the iron against a flat part of the joint, but heat transfer is still sometimes too low. Our solution is to hold the thin strand of solder between the joint and the iron until it melts, and then move the solder to the opposite side. The tiny bit of solder which melts on the iron's side of the joint forms a layer of metal which transfers heat very quickly to the joint, so that by the time we get the rest of the solder to the other side, the joint is hot and ready to melt more solder. In this way we can solder a connection on a pc board in just a second or two.

Speed is important too. Applied too long, the soldering iron can burn the board or loosen a trace so it comes off; both of these are unsightly and can be difficult to fix. Some companies sell heat sinks - small clips which are supposed to remove extra heat from connections - and many people suggest putting a rubber band around the handles of a pair of needle-nose pliers so they stay closed, and then clipping them on the lead being soldered. But these do not really solve the basic problem; if anything, they make it worse because they keep the temperature too low for good soldering. The best solution is to keep the iron clean and hot so that the connection can be made fast - if a single printed circuit board connection takes you more than a second or two, you are doing something wrong.

When finished, the connection should be smooth, bright and shiny, with no rough edges. Don't use too much solder - the right amount will just cover the joint. In fact, it is not even essential to cover all of the joint, as long as there is enough solder to form a bridge between the two metals to be joined. Many technicians believe that it is better to have too little solder than too much.

Note that both sides of the board are covered with what looks like a layer of thin green paint. This is called the solder mask. The entire side of the board is masked except right around each hole; the purpose of the solder mask is to keep the solder on a joint from spreading to adjacent joints. Since many of the solder pads are very tiny, the solder mask helps prevent solder bridges from shorting to nearby pads. All exposed areas of copper are plated with a stable, shiny coating which does not corrode. Unlike plain copper boards, which slowly oxidize and have to be washed or cleaned just before soldering for best connections, this board does not need such treatment. Do not wash or clean the board prior to soldering.

Final words - don't rely on the above. If you have never soldered to a printed circuit board before, find someone who has and ask them to show you.

Appendix D

Disk Organization

The data on a floppy disk is written in cylindrical paths called tracks. The most common disk format is to have a 5¼" disk with 40 tracks, spaced 1⁄48" apart; this format is often called a 40-track or 48 TPI (tracks per inch) disk. A somewhat newer format places the tracks half as far apart, resulting in 80 tracks which are 1⁄96" apart; this is usually called a 96 TPI disk. 3½" disks also come in 40- and 80-track versions, but their tracks are placed considerably closer together. Tracks are usually numbered starting at track 0, which is the outermost track, to track 39 or 79, which is the innermost track.

In addition, disks can be either single-sided or double-sided, although almost all modern disk drives write on both sides. Obviously, a double-sided disk can store twice as much data. Disks are often described as SS for single-sided or DS for double-sided.

Furthermore, disks can be either single-density, double-density, or quad-density, depending on how closely the bits are packed together on a track. The original floppy disks were all single-density, but the more modern double- and quad-density formats pack roughly twice as many, or four times as many, bits on a track (although as a practical matter, the actual improvement is usually about 20% less than expected.) Densities are often abbreviated as SD for single density, or DD for double density.

Finally, most floppy disks rotate at 300 rpm, which works out to 5 revolutions per second, although the so-called HD or high-density disks on IBM computers rotate at 360 rpm (the same speed, in fact, as the really old 8" disks.)

Since many people are familiar with the various disk formats available on an IBM personal computer, here is a comparison of some of these common formats:

Capacity (bytes)	Sides	Tracks	Density	RPM	Size
180K	1	40	Double	300	5¼"
360K	2	40	Double	300	5¼"
720K	2	80	Double	300	3½"
1.2 meg	2	80	High	360	3½"
1.44 meg	2	80	High	360	3½"

The original SK68K computer supported only the double-density formats in the above table; now it supports all of the above disk formats (although older versions require a newer HUMBUG ROM, and a plug-in IBM-style disk controller).

Although the typical double-density track can theoretically hold almost 6000 bytes, in practice only 4608 bytes are used for actual data storage; the remaining bytes are wasted. On an IBM disk running MS-DOS, these 4608 bytes are divided into 9 sectors of 512 bytes each; on the SK68K running the SK*DOS disk operating system, they are divided into 18 sectors of 256 bytes each. These sectors are then numbered, beginning with sector 1 at the beginning of a track.

On an SK*DOS disk, track 0 is used to hold system information, while the remaining tracks hold program and data files. On track 0, sectors 1 and 2 hold the *superboot* program which is used when starting (booting) the system; sector 3 is called the *System Information Sector* (SIS) because it stores system information such as how much of the disk is free; sector 4 is used for testing purposes, and the remaining sectors, beginning with sector 5, hold the disk directory; if more space is needed for the directory, then it may be continued on other tracks.

Each file on the disk has an entry in the directory which contains the file's name, size, time and date of creation or last update, location on the disk, and a one-byte attribute which provides further file information. The location of the file is specified by the track and sector where the file begins, and the track and sector where the file ends.

Since only the beginning and ending locations are specified in the directory, additional information which tells where to find the rest of the file is contained within the file itself. The first two bytes of every sector in the file contain a pointer to the next sector of the file, in the form of another track and sector number. Since each sector therefore points to the next, the sectors in each file form a *chain*; this type of disk organization is thus called a *linked chain* system.

The free space on the disk is treated as another linked chain of sectors, whose beginning and ending locations, as well as size, are stored in the SIS, sector 3 of track 0. As the SK*DOS disk operating system creates or deletes files, it simply moves sectors between chains. For example, when a file is deleted, its name is removed from the directory, and its sectors are added to the end of the chain of free sectors. One neat side-effect of this system is that, if there is enough free space on the disk, these sectors may not be overwritten for some time. It is therefore possible to pull back a deleted file, sometimes even days or weeks later. SK*DOS is supplied with an UN-DELETE program which does exactly that.