

---

# 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  $\overline{\text{LDS}}$  and  $\overline{\text{UDS}}$ .

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.

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

$\overline{\text{IPL0}}$ ,  $\overline{\text{IPL1}}$ , and  $\overline{\text{IPL2}}$  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.

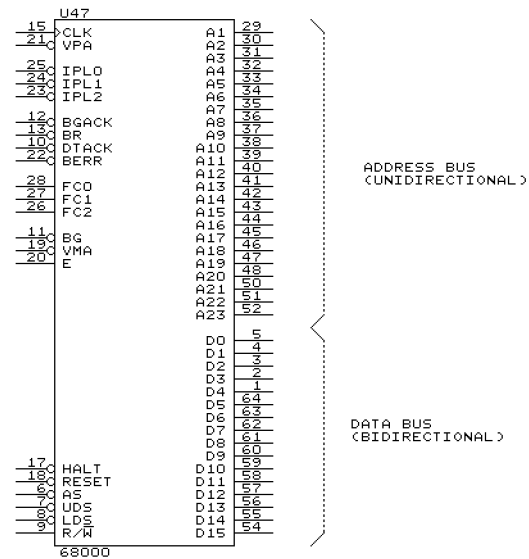


Fig. 6-1. 68000 microprocessor pinout.

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 ( $\overline{\text{BG}}$ ). The DMA controller would then send a Bus Grant Acknowledge ( $\overline{\text{BGACK}}$ ) signal to confirm that it has control of the busses, and temporarily take over the system while the 68000 sits back and waits.

$\overline{\text{LDS}}$  and  $\overline{\text{UDS}}$  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  $R/\overline{W}$  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  $R/\overline{W}$  is high) or write data out (when  $R/\overline{W}$  is low). Thus  $R/\overline{W}$  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.

$\overline{BERR}$  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,  $\overline{DTACK}$  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  $R/\overline{W}$ , (c) outputs the address strobe and  $\overline{LDS}$  and/or  $\overline{UDS}$ , and then sits back and waits. It waits until it either gets back  $\overline{DTACK}$ , indicating that the transfer is finished, or  $\overline{BERR}$ , indicating that something went wrong. When  $\overline{DTACK}$  is received, then the 68000 goes on to the next step.

If  $\overline{DTACK}$  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,  $\overline{DTACK}$  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,  $\overline{DTACK}$  can be delayed so the 68000 waits for it to finish.

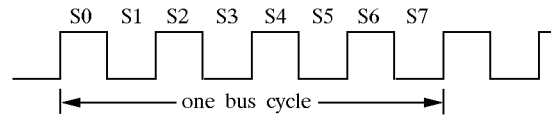
In practice, each 68000 memory or I/O access takes a certain minimum amount of time, measured in cycles of the MPUCLK signal. If  $\overline{DTACK}$  is delayed, even just an instant, the 68000 lets an entire extra clock cycle slip by and checks again. If  $\overline{DTACK}$  is still off, then the 68000 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 ( $\mu$ s). (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

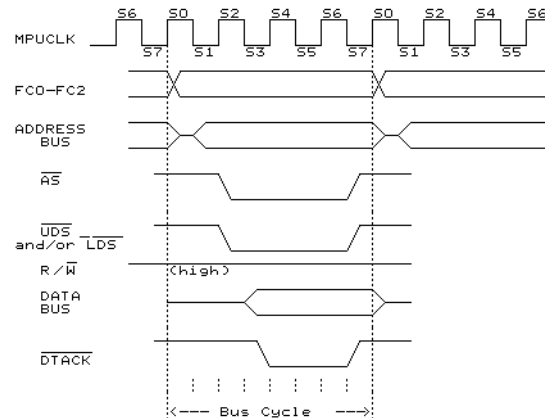


Fig. 6-2. Normal read cycle timing

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  $\overline{UDS}$  and/or  $\overline{LDS}$ . It asserts  $\overline{UDS}$  for a byte read from an even address,  $\overline{LDS}$  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  $\overline{AS}$ ,  $\overline{UDS}$ ,  $\overline{LDS}$ , 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  $\overline{DTACK}$  to tell the 68000 that the data is available. In order to continue as shown, the 68000 needs  $\overline{DTACK}$  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.  $\overline{AS}$ , and  $\overline{UDS}$  and/or  $\overline{LDS}$  are negated (they go back high).

Once  $\overline{AS}$ , and  $\overline{UDS}$  and/or  $\overline{LDS}$  go off, the memory or I/O device is supposed to turn off  $\overline{DTACK}$  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,  $\overline{DTACK}$  is supposed to arrive before the end of state 4. What happens if  $\overline{DTACK}$  is delayed (by slow memory, for example)? Fig. 6-3 shows what happens.

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

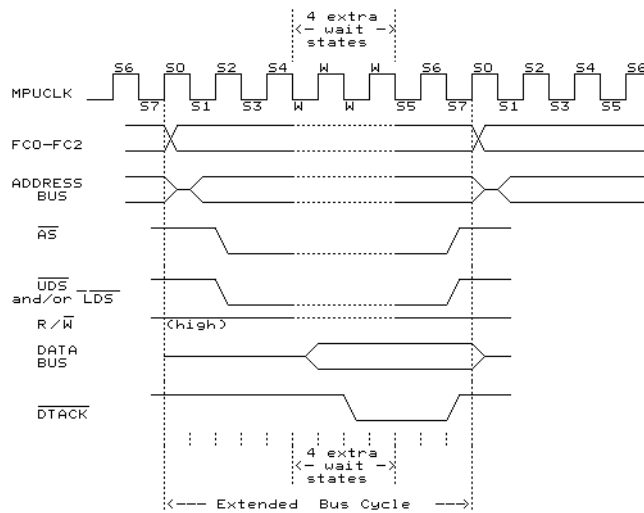


Fig. 6-3. Extended read cycle timing.

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

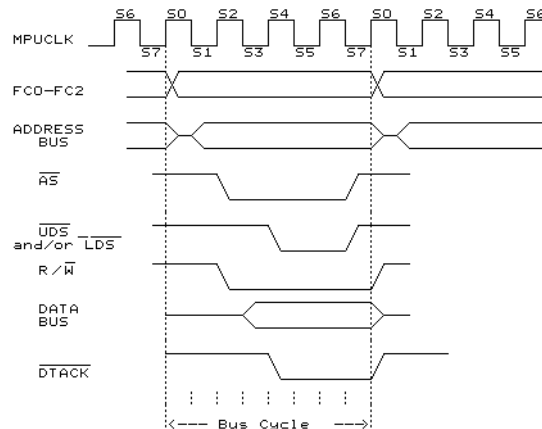


Fig. 6-4. Normal write cycle timing.

2.  $\overline{UDS}$  and  $\overline{LDS}$  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  $\overline{LDS}$  or  $\overline{UDS}$ .

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-

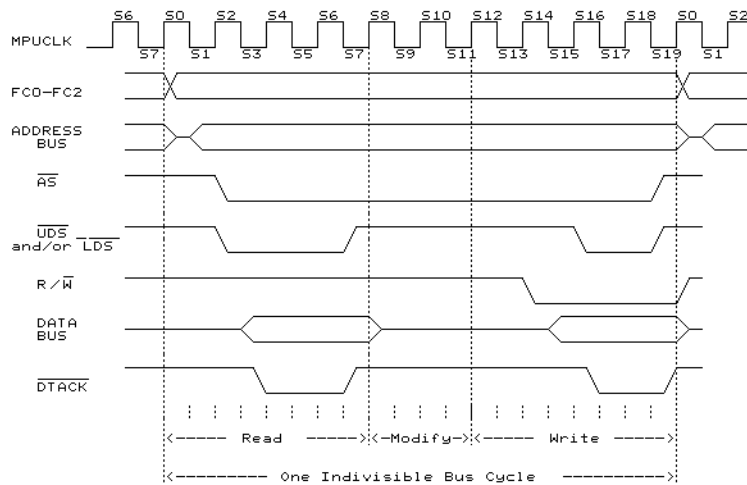


Fig. 6-5. Read-modify-write cycle timing.

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

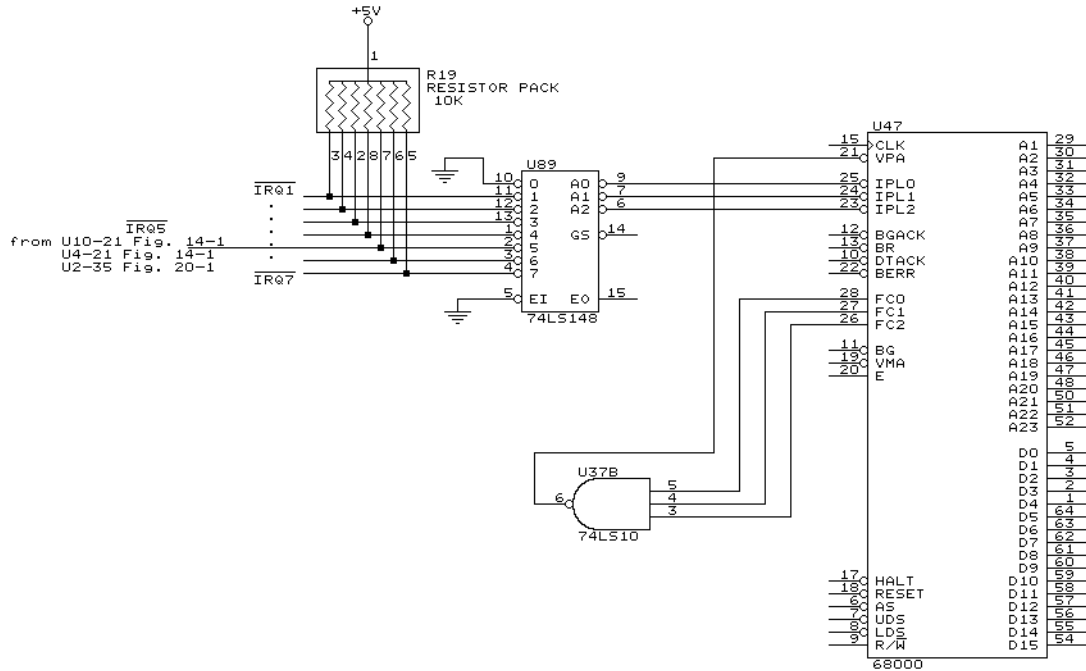


Fig. 6-6. 68000 Interrupt circuitry.

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.

