# Troubleshooters' Guide

*You're hesitant about tackling repair and interfacing problems? If so, this will point you in the proper direction and get you started.*

*Ralph Tenny*
*P O Box 545*
*Richardson TX 75080*

One of the fascinating phenomena about computers is that they can do things—almost anything—*automatically*, provided the machinery to accomplish a given task is available and can be run and controlled by electrical signals. All such computer-controlled machines are called peripherals, no matter what their function is. Most of us routinely use such peripherals as TTYs or TVTs, audio cassettes and printers. Depending on whether you bought a system or built kits, you spent various amounts of time getting those peripherals to work with your computer.

You were probably furnished detailed instructions for operating the TTY or TVT and cassette with your computer, and the necessary electrical connections (interface circuitry) were already designed and ready to use. Finally, it is almost certain that the software for your computer already had provisions to operate the peripherals necessary to make the computer functional. These things are necessary for any computer system—interface, software and "how to"—but the information may not always be available when a peripheral made by one manufacturer is to be used with a computer from another manufacturer. In the case of surplus equipment such as a Baudot TTY, there may be no instructions or software available.

### The Big Picture

Regardless of the circumstances, let's assume you are having trouble with a computer peripheral (otherwise why are you reading this?). The troubleshooting approach needed will vary with the type of computer and hardware that's involved.

The computers available to most hobbyists will be one of two types—those with isolated, or accumulator, input/output (I/O) such as the 8080 and 2650, or those with memory-mapped I/O such as the 6800 and 6502. Accumulator I/O machines have special input and output instructions, while the memory-mapped computers use standard memory instructions such as LOAD and STORE to service both memory and peripherals. In order for this to happen, such peripherals are assigned memory addresses; this will limit the total amount of memory available. However, since most microcomputers will address either 32K or 64K words of memory, it would take a lot of peripherals to make a dent in the available memory space!

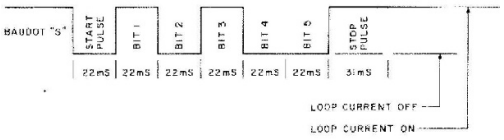


*Fig. 1. Baudot code timing.*

| HEX CHARACTER | BAUDOT | HEX CHARACTER | BAUDOT |
|---|---|---|---|
| 0 | 01101 | 8 | 01100 |
| 1 | 11101 | 9 | 00011 |
| 2 | 11001 | A | 11000 |
| 3 | 10000 | B | 10011 |
| 4 | 01010 | C | 01110 |
| 5 | 00001 | D | 10010 |
| 6 | 10101 | E | 10000 |
| 7 | 11100 | F | 10110 |
| CARRIAGE RETURN | 00010 | LINE FEED | 01000 |
| SPACE | 00100 | LETTERS | 11111 |
| FIGURES | 11011 | | |

*Fig. 2. Chart of some Baudot codes.*

The best troubleshooting method also depends upon which type of peripheral is involved—whether it is a device to input or output data or if it controls something. In general, controllers are a bit easier to troubleshoot because their input signals (combinations of bits on the input lines) are less numerous than those for data-handling devices. Also, it is likely that the interface connections will be simpler for the controller than for the data peripheral.

The final consideration will be software; detailed instructions must be available to enable the computer to produce the proper signals to drive any peripheral. Most manufacturers of hobbyist equipment furnish software for peripherals they produce. If surplus equipment is involved, there may be no software available unless someone has also made a kit available to interface the machine to a microcomputer. Note that if appropriate software is available, it will depend strictly upon a properly functioning interface of a particular design.

Successful troubleshooting of computer peripherals requires careful study of the system—from microcomputer architecture through the mechanical and electrical details of the peripheral and interface. You are probably familiar with your computer, so let's begin with the peripheral, giving it a thorough inspection to be sure it is totally functional. Operate all controls, and supply electrical signals if appropriate. Proceed to the interface circuitry only if these tests are successful.

Be sure the interface is capable of reliably and safely producing all the signals needed by the peripheral. Relay contacts or semiconductor switches (power transistors, SCRs or Triacs) must be able to handle the voltages and currents involved. A stuck relay or defec-

tive Triac won't hack it—all systems must be GO! Power supplies must be able to handle the load. If the computer power supply is used, be sure there is reserve capacity and that the computer power bus isn't receiving power glitches caused by the peripheral.

Now, consider the software. Let's take the case where the software is for a different computer system. Break it down into modules so that only one peripheral operation at a time is addressed. Determine what changes must be made to perform the same function on your computer, and make changes as necessary. Finally, combine the modified modules to check out the entire system—computer port, interface and peripheral—in a simple loop or repetitive fashion that addresses all desired peripheral functions. At this point, you should have mastered the system well enough so you can create application software to make the peripheral do useful tasks as a part of the whole system.

**Serial Data Testing**

Now, let's examine an actual problem. My KIM-1 has software (monitor ROM) and an interface to drive an ASR-33 Teletype directly; but how about operating my Model 15 Baudot TTY? All I have is documentation for KIM and a service manual for the Model 15—no software and no interface circuitry. I want the Model 15 to serve only as a printer, so there is no need to interface the keyboard; KIM would drive the printer mechanism using whatever data I wish to feed it. The keyboard was useful in checking out the machine initially, but the interface will be much simpler if the keyboard is not involved.

The first step in planning this project is to understand how the machine works. The print mechanism operates when current through a selector magnet is interrupted in a certain code pattern. Fig. 1 is a timing diagram of the Baudot code format, and Fig. 2 is a chart of Baudot codes for the hexadecimal and TTY control characters needed. Note that a Baudot machine has no SHIFT key, but that it has LETters and FIGures keys. (The five-level code will select only 32 keys, but by arranging for numbers, symbols and punctuation marks to be uppercase—FIGures mode— the print set is expanded to 58 codes.)

Look again at Fig.1 and note that the Baudot code consists of a start bit, five code bits and an extra-length stop bit. That stop bit could be troublesome to make, so let's modify the code format as shown in Fig. 3—that is, substitute two regular-length stop bits for the longer one. Previously (Fig. 1), the character time was 163 ms; now it is 176 ms—only 8 percent slower. The resulting simplification of software and hardware makes the trade-off entirely acceptable.

Now, how can KIM drive the Model 15? KIM has a 20 mA current loop derived from a 5 V supply, which is entirely inadequate to switch the 60 mA current derived from the high-voltage supply of the Model 15 (Fig. 4). Of course, it is possible to add some external circuitry to adapt KIM to Model 15, but KIM's software is still a problem. KIM's lookup table is for hexadecimal to ASCII, and the self-adjusting timer, which produces the proper output bit rate, requires the keyboard for setup. It is easier to build a simple interface for the Model 15 and drive it from the standard KIM output lines.

Fig. 5 is a simple CMOS circuit that accepts parallel data and will shift it out to the Model 15 through two transistors. Q1 matches the CMOS output to the higher drive requirements of Q2, which is a high-voltage unit to switch the current loop. IC1 is a shift register, and IC2A is a free-running oscillator with a period of 22 ms. IC2B is a pulse generator that causes the shift register to load data from the five input switches.

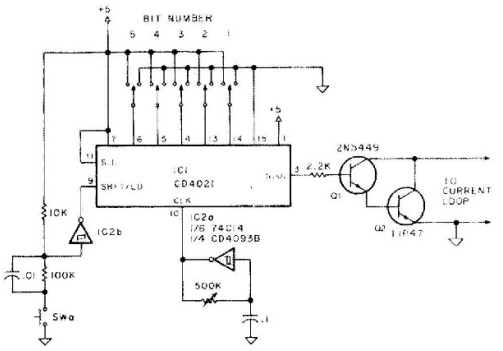


*Fig. 5. Parallel-to-serial driver for Model 15 TTY (with test switches).*

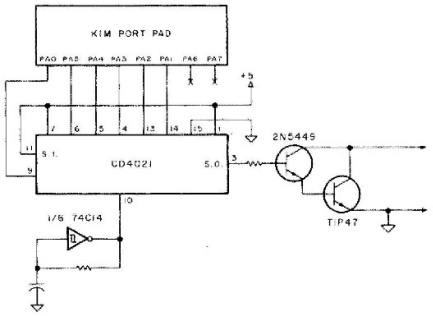


*Fig. 6. Completed Model 15 interface.*

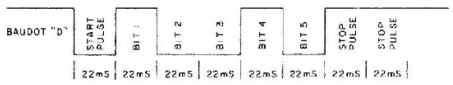


*Fig. 3. Modified Baudot code timing.*

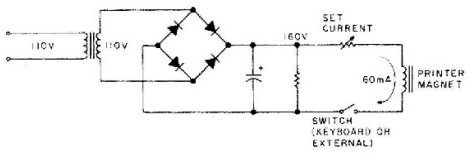


*Fig. 4. Common Model 15 current-loop supply.*

A cycle of operation works this way: push-button switch SwA trips IC2B, and data is loaded into IC1, including a 1 on pins 1 and 7 and 0 on pin 15. As soon as the load pulse terminates, data is ready to shift out as clocked by IC1A. When the 0 loaded by pin 15 reaches the output, Q1 and Q2 turn off, producing a start pulse for the Model 15. As the remainder of the data is shifted out, 1s are shifted in via pin 11, the serial input pin. Thus, after loaded data is shifted out, Q1 and Q2 remain turned on, waiting for the next data to be loaded.

The proper test method for this interface is to set the switches to the Baudot codes for R (01010) and Y (10101) alternately. Carefully adjust the frequency of IC2A until proper operation is obtained, then try other characters. The circuit in

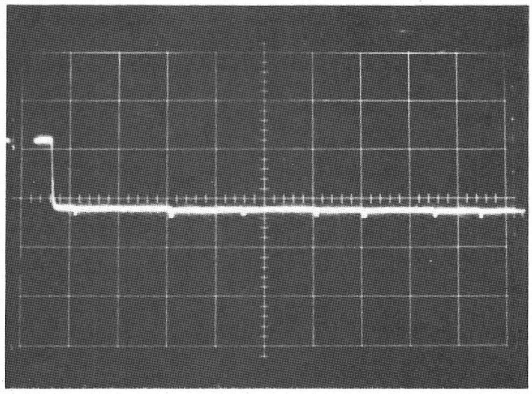

*Photo 1. Instruction (LDA 8000) causes Address Bit A15 to go high as the instruction is executed; this defines a unique strobe to furnish sync for an oscilloscope.*

| Label | Op. | Arg. | Comments |
|---|---|---|---|
| TEST | LDA | #$3F | Select mask for port. |
| | STA | PADD | Set mask in port control register. |
| | LDX | #$00 | Zero index register as counter. |
| PRINT | LDA | BUFFER, X | Get first data byte. |
| | STA | PAD | Set data in port. |
| STROBE | INC | PAD | Toggle PA0 to load data |
| | DEC | PAD | into interface. |
| | INX | | Set index for next data byte. |
| | CPX | END | Test for last byte. |
| | BEQ | OUT | Done? If so, exit. |
| WAIT | LDA | #$F0 | Set time in |
| | STA | TIMER | programmable timer. |
| TIME | BIT | TIMER | Timer done? |
| | BPL | TIME | No, go back and check again. |
| | BMI | PRINT | Yes, print again. |
| OUT | BRK | | Stop computer after printing last byte. |

*Fig. 8. Short test program writes data from location BUFFER to interface.*

Fig. 5 has now become a tested interface for the Model 15.

Although it was designed with KIM in mind, it can easily be adapted to any computer. Fig. 6 shows the KIM output port attached to the interface; note that PA0 (least significant bit of the port) replaces IC2B. Data can be written out to the port on pins PA1 through PA5, and the PA0 can be toggled (turned on and off) to load the data.

Software to operate the interface can take many forms, depending upon how the printer needs to work with a main program. For checkout, a short test routine is best. Fig. 7 is a flowchart and Fig. 8 is the KIM program. Note how the software is intimately related to the hardware.

Begin with the pin assignments of the output port: PA0 is the least significant bit (LSB), so it can be toggled with INCrement and DECrement instructions. The data buffer has the organization shown in Fig. 9, where X stands for "don't care," or unused, bits. Bit 0 (LSB) is always 0, so the load/shift pin of IC1 is always low for shifting until new data is to be loaded.

Let's follow through the program after a brief comment on parts of the setup. At label TEST, 3F is loaded to the port. Ones loaded into the Data Direction register (PADD) turn the corresponding port lines into outputs, while 0s create inputs. So 3F makes bits PA0 through PA5 outputs as required for the interface. Output data is stored at a group of addresses named BUFFER, and the location named END contains data specifying how many words of data BUFFER contains.

The location named TIMER is a programmable timer that sets bit PB7 low when time is up. So, when the progam is entered at TEST, the port is set up and register X is zeroed to make a counter. Data is loaded at PRINT, stored, and then IC1 is loaded by STROBE. Register X is incremented and tested; the routine at WAIT marks time until IC1 completes a print cycle.

If the last byte has been printed, the BRK instruction at OUT stops the computer. By now, it is even more apparent that software and hardware must work in exact harmony if any computer-controlled task is to be successfully completed!

The example above may seem contrived and simple, but it illustrates the most important points about debugging peripherals and their interface circuitry: *Never* try to debug malfunctioning equipment with an applications program. Always break up the task into as many modules as possible. It's OK to fire up a peripheral with furnished software if and *only* if you are dealing with a turnkey package in which the software and hardware were created for each other. *Then* if it doesn't run, follow the suggestions and examples above.

Let's return to the point where manual and electrical testing of a peripheral seems to prove the mechanism functional. I will assume that you understand the signals and power that must be furnished by the interface. If you don't, stop until you find out! If the interface furnishes power, substitute a similar load and write a

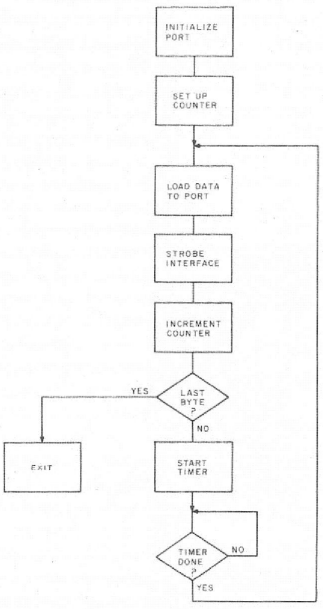


*Fig. 7. Flowchart for test program.*

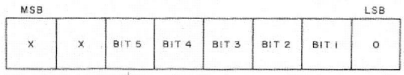


*Fig. 9. Data buffer organization for loading Model 15 interface.*

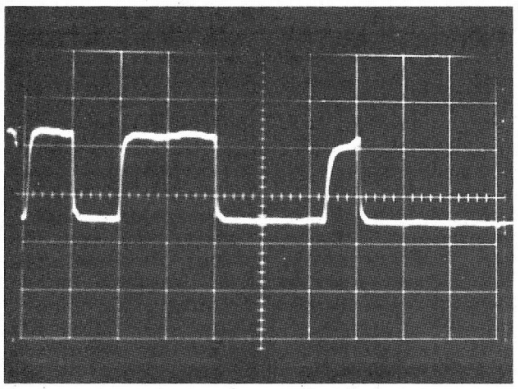

*Photo 2. Using the sync shown in Photo 1, Data Bit 2 is examined for proper data activity and timing (see text for details).*

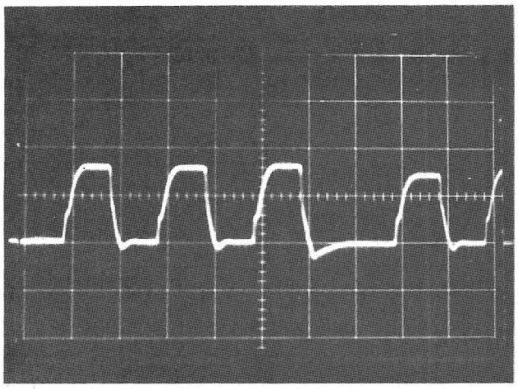

*Photo 3. Heavy capacitative loading of buffered address line produces slow-rising leading edges and undershoot on trailing edges.*

short program that turns on the power and then halts the computer (or loops while waiting for a Reset signal). Measure the voltage across the substituted load to be certain power is really being delivered. Check for switch closures with an ohmmeter. Do each function of the interface the same way until it all checks out.

**Scope Techniques**

Data transfers or special signal patterns are harder to check. The *neat* way to do this is to use a digital analyzer—about $2K worth. Not too many of us have one, so there must be another way. The next easiest way to troubleshoot is to use an oscilloscope. Due to the short time any particular word of data stays on the data bus—just one part of a machine cycle, which may be less than a microsecond—the scope must have a triggered sweep. Even then, it is difficult to find a trigger signal that triggers only when (or just before) the data to be checked appears on the data bus. It is sometimes possible to make the computer generate a unique strobe shortly before outputting a data byte. For example, Fig. 10 is a short program that generates a unique address (LDA 8000 sets bit A15 high) and then moves data to the address bus, generates a short delay and repeats. When the scope is triggered by bit A15 going high, it is possible to examine the data bus bit by bit and verify that the proper data is appearing.

Photo 1 shows bit A15 of the address bus going high at the first statement of the program in Fig. 10 (LDA #8000). When this signal is used as a trigger for the scope, it is possible to examine the data bus and watch for the data to appear. This is shown in Photo 2. The exact details of what follows will be pertinent only to the MCS 6502, which is the processor used in the KIM-1 microcomputer, and a similar analysis will have to be made for each different uP.

| Label | Op. | Arg. | Comments |
|---|---|---|---|
| TRIG | LDA | $8000 | Set address bit A15 high as strobe. |
| | LDA | #$A5 | Get data for output. |
| | STA | PORT | Send data to port. |
| | LDX | #$F0 | Set up index register as counter. |
| COUNT | DEX | | Decrement counter. |
| | BNE | COUNT | Loop back if counter not zero. |
| | BEQ | TRIG | Start over after counter reaches zero. |

*Fig. 10. Short program generates scope trigger to verify data movement.*

In Photos 1 and 2, the timebase speed is such that one machine cycle takes one horizontal division on the scope face. Keep that in mind, and it will then be possible to analyze when the data should appear on the data bus. The next step is to count machine cycles through the program (see Example 1).

Since the absolute address appears only in the last cycle of the first instruction, Photo 1 shows that ending cycle. Counting forward six more cycles (six divisions on the scope graticule in Photo 2), we see that a data bit comes high in that cycle. The subject of Photo 2 is Data Bit line 2, which should be a 1 according to the data loaded in instruction 2. The bit pattern with a data byte of A5 is shown in Example 2. Thus we see that Data Bit 2 *should* be a 1; checking other bits on the data bus showed that the correct data was appearing at each pin.

To summarize the procedure for checking data (or address) bit on computer bus lines, begin by creating a software strobe or other means of sync for the scope. Set the scope time base so that one machine cycle occupies one horizontal division of the scope graticule. Analyze the program to determine the number of machine cycles required to bring the data to the bus; remember that this analysis depends upon

| Instruction | Type Instruction | Cycles |
|---|---|---|
| 1. LDA $8000 | Load accumulator absolute | 4 (use last cycle only) |
| 2. LDA #$A5 | Load accumulator immediate | 2 |
| 3. STA Port | Store accumulator absolute | 4 |
| | | 7 |

*Example 1.*

| Data Bit | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 |
|---|---|---|---|---|---|---|---|---|
| Binary value | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |

*Example 2.*

| | Address lines | A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Address | 10AF | X | X | O | N | X | O | O | X | N | O | N | O | N | N | N | N |
| | 0F36 | X | X | O | O | N | N | N | N | O | O | N | N | O | N | N | O |
| | 3668 | X | X | N | N | O | N | N | O | O | N | N | O | N | O | O | X |

*Example 3.*

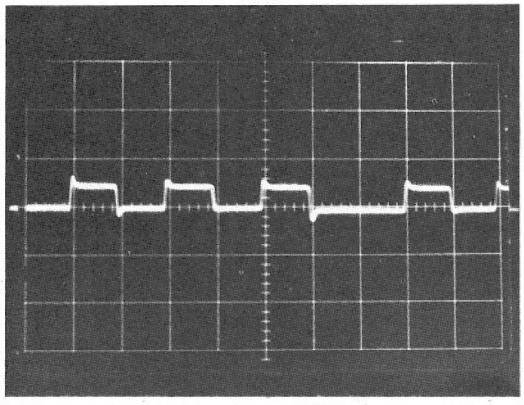

*Photo 4. High resistive loading (partial short to ground) reduces address line amplitude to below logic 0 levels.*

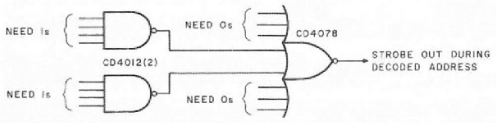


*Fig. 11. Selective decoding gives unique event to check program branching.*

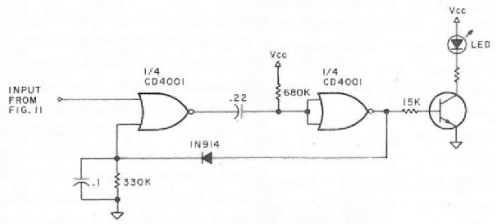


*Fig. 12. Simple CMOS one-shot makes LED blink slowly from repetitive triggers.*

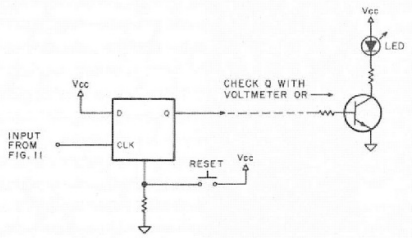


*Fig. 13. Flip-flop records infrequent triggers.*

thorough knowledge of the machine timing and the number of cycles each instruction takes. Finally, determine where (on the scope graticule) the data should appear, and look at each line on the bus to verify correct operation.

Photo 3 shows a buffered address line that has heavy capacitive loading. Note that rising edges are heavily rounded, and that falling edges have undershoot with a slow recovery. Photo 4 shows the same address line with a heavy resistive load; note that the amplitude is drastically reduced and obviously will not meet the voltage levels required to operate either TTL or MOS circuits. Any device driven by the line shown in Photo 3 will probably show erratic or false address decoding; the waveform of Photo 4 will probably cause a driven device to consider this address bit a 0 at all times. In either case, faulty addressing will be the symptom and a scope would be needed for proper diagnosis.

### Other Tools

It is often possible to use ingenuity and planning to do much troubleshooting with a voltmeter or other static indicator. Also, if your system has a front panel with address switches and status lights, the pattern on the lights may offer helpful hints. If data fails to appear at the expected place, it can be very helpful to know that the subroutine that moves the data wasn't called by the main program. How? If the subroutine happens to be in a little-used page of memory, sometimes it is possible to see the address LEDs on the front panel flicker as the subroutine is accessed. A program loop can be used to enhance the brightness of the LEDs.

In a similar vein, checking certain address bits with a logic probe (a "pulse catching" feature is necessary) can reveal that the computer is accessing certain parts of memory. If no single unique address bit is involved, a simple two-IC circuit (Fig. 11) will decode enough of the address to generate a unique pulse each time the memory accesses the decoded address. This pulse will then trigger the logic probe so that it blinks. If a logic probe is not available, hook up a one-shot (Fig. 12), which will make a blinking light, and trigger it from the decoder of Fig. 11.

The decoder of Fig. 11 works this way: It can decode 14 address lines, but the choice of these lines will depend both on the address to be decoded and on other parts of the program with similar addresses to be excluded. Example 3 shows some hex addresses and possible decoding connection choices. N represents a NAND input line, O is a NOR connection, and X indicates lines left open.

The object is to make the best use of the available gate inputs so that only addresses within the subroutine are decoded, while no addresses in the main program are decoded. This will ensure that the decoder will develop an output only when the subroutine is addressed.

Finally, a simple flip-flop (Fig. 13) can be triggered by the address decoder. This is particularly useful for checking on events that happen infrequently, such as monitoring switch closures. Select an address within the program section that reacts to the switch closure; then close and open the switch. If the flip-flop is set by the decoder, all is well. Use a voltmeter to check the Q output of the flip-flop, or use a transistor driver to turn on an LED. Check both states—be sure the flip-flop is set by the computer's response to the switch closure and that it stays reset as long as the switch stays open. ■