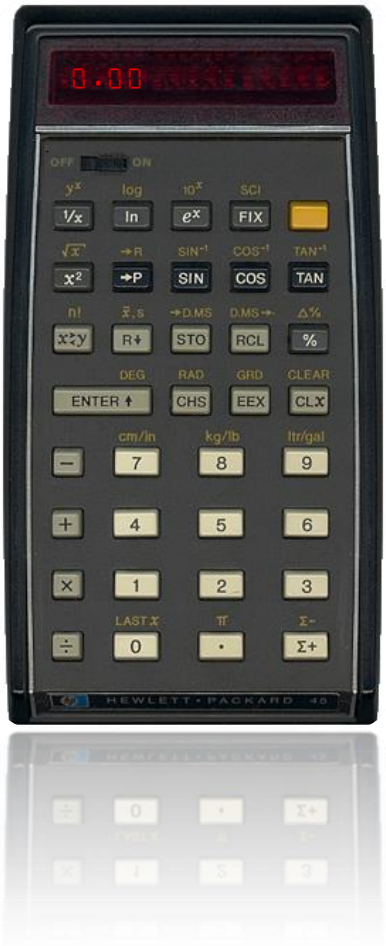
Notes on HP’s Classic Calculators



T Nixon 2022

Contents

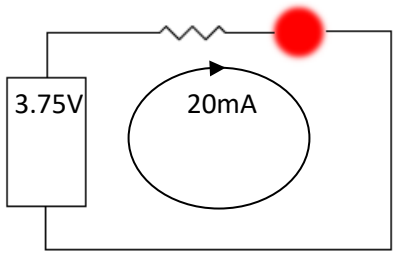
Light Emitting Diodes (LEDs) are a great and easy way to display numerical or alphanumerical information.

LEDs require current passing through them to create light. This current is usually in the 10 to 20 millamp (mA) range and when this current flows, the LEDs will develop a voltage drop across them. For red LEDs, this voltage is about 1.7 volts. Most circuits that have LEDs are powered by a voltage higher than this and if you try to drive a LED directly from this higher voltage it will be damaged from too much current flowing through it. One common method to reduce the current is to place a resistor in series with the LED.

If we decide that the LED requires 20mA to be bright enough for an application powered by a 3.75 volt supply then by Ohms Law we can decide the resistor value. If the LED drops 1.7 volts, then the resistor will have 2.05 volts across it. (3.75 – 1.7 = 2.05) The current flowing through the resistor generates a small amount of heat which could be considered a waste of energy.

$$R = \frac{V}{I} \quad \frac{2.05}{0.02} = 103 \text{ ohms}$$

103 ohms



As you can see, this simple LED circuit consumes little power, but If you consider a 15 digit, 7 segment LED display which could have up to 87 LEDs on at any time then the energy consumed becomes a problem because the battery will not last long trying to drive that sort of load. (87 x 20mA is 1.74 amps)

It was found that supplying a short duration pulse of high current at regular intervals could also drive the LEDs at the required brightness. This current could be in the 100's of milliamps, but is only for very short periods of time and does not cause any detrimental effects on the LED. In fact by using this method, the LEDs are mostly turned off. With some clever electronic design, a multiple LED display could be driven like this and is how the Classic display works.

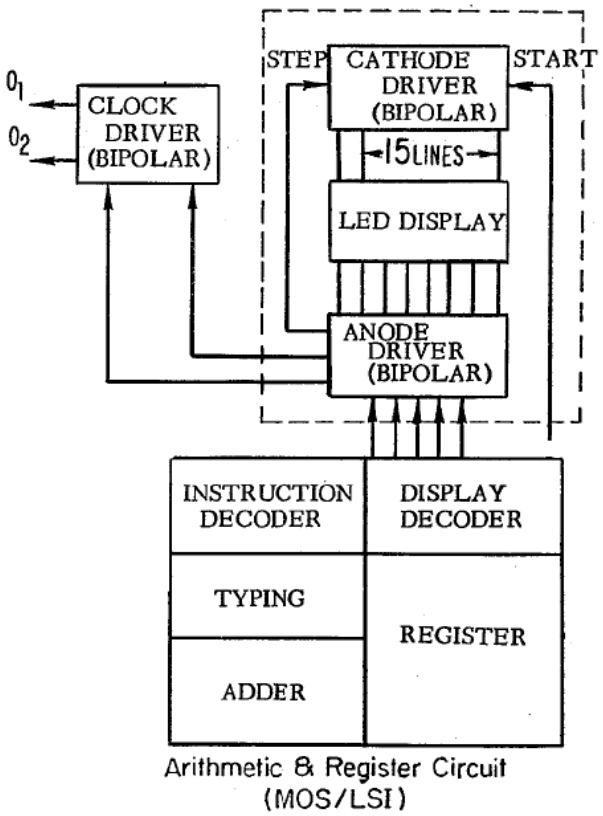
It takes 280uS (millionths of a second) to refresh the display and out of this, each LED is on for about 5uS. In other words, each LED is turned on for approximately $1/56^{th}$ of the display refresh time. The refresh process that controls which LEDs are on or off happens so fast that the human eye cannot detect any flicker. The result is a bright display with a much smaller amount of power being used to drive it.

Further power savings were realised by charging small inductors and then discharging them into the LEDs to light them. This meant that the series resistors were not required and thus the power they consumed was eliminated. Another feature is that the displays are powered directly from the battery which means the power supply circuit does not have to be beefed up enough to supply the extra power.

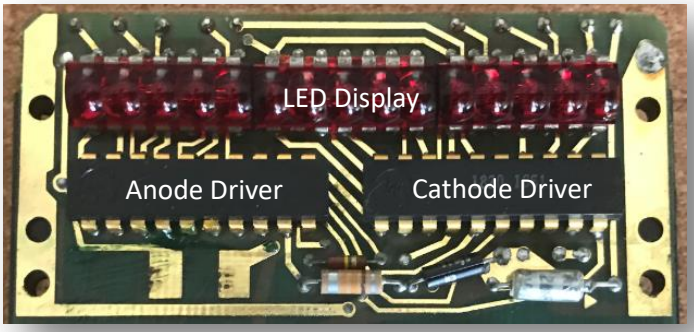
Two specialist integrated circuits were developed to drive the display in this manner. One is called the Anode Driver and the other is the Cathode Driver. These match the connections to the LEDs which also have an anode and a cathode.

The cathode driver sequentially activates each of the 15 display digits one at a time. It can do this because the cathodes of each display LED are connected together. The anode driver then sequentially turns on each of the appropriate LED anodes for that digit. Each digit is active for 20uS and therefore a full display refresh takes 280uS. (But 15 digits = 300uS ???) This anomaly is due to the fact that while the decimal point is displayed in a separate digit, it is actually lit up during the same 20uS for the digit that it follows.

The Arithmetic & Register Circuit (ARC) has 5 data lines connected to the anode driver. These lines labelled A, B, C, D and E transfer partially decoded display information. The anode driver fully decodes this information into eight output lines which connect to the 7 LED segments of each display and one to the decimal point. The reason for the partial decoding is to help reduce power consumption and probably simpler design.



Arithmetic & Register Circuit (MOS/LSI)



On/Off switch contacts 800KHz Clock Timing

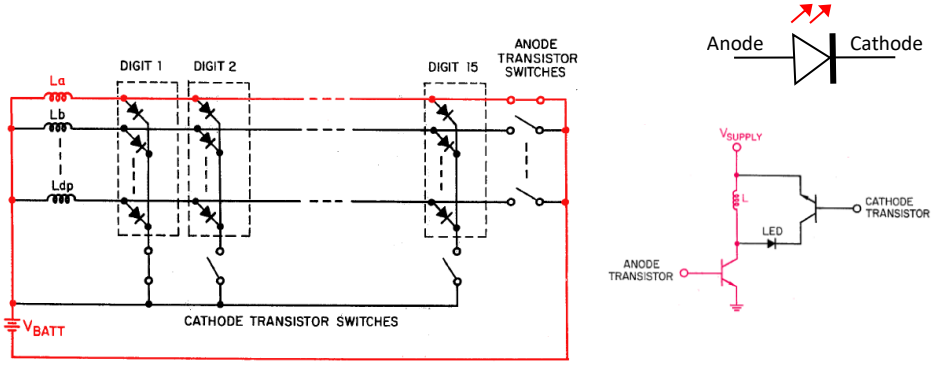

HP-55 ARC Chip

Underside of circuit board

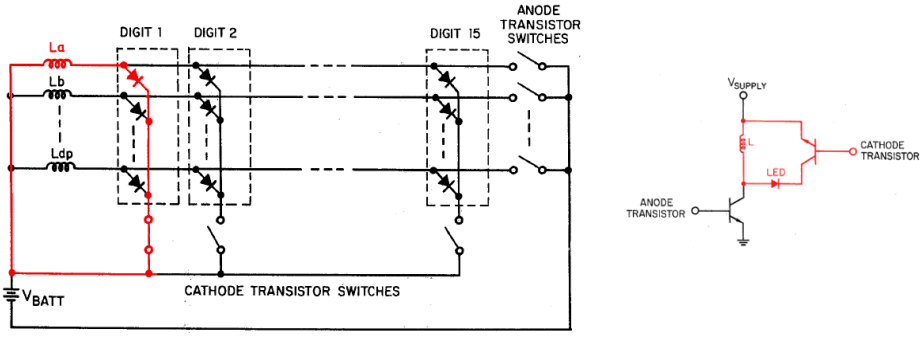
The quad inductor packages have different colours because of the decimal point LED. The 7 LED segments of each digit require one 130uH inductor each to store the charge. The decimal point LED being lit for half the time of the others, only requires a 68uH inductor. Therefore one quad package has 4 x 130uH inductors and the other has 3 x 130uH and 1 x 68uH inductors*.

ELECTRICAL PATHS FOR THE LED DISPLAYS

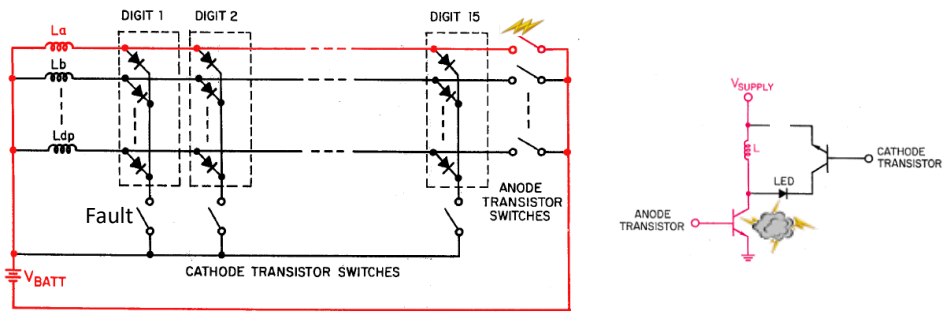
In the following diagram, the anode switch for LED segment (a) is closed. The cathode driver has selected Digit 1 to display information. The inductor (La) charges from the battery, through the anode switch and back to battery again. The charge process is allowed to continue for 2.5uS for LED segments a – g, and 1.25uS for the decimal point.



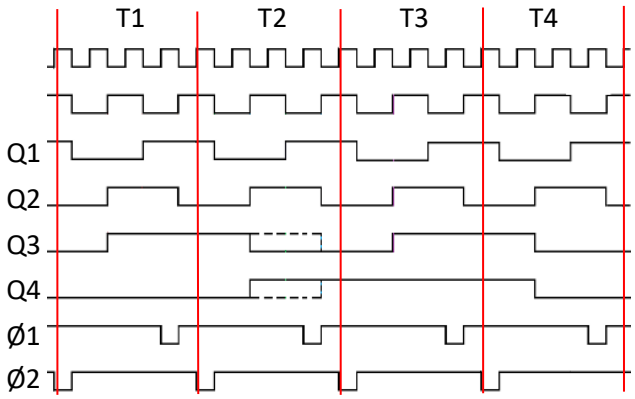
After the time interval, the anode switch opens and the cathode switch stays closed. The inductor stops charging and now discharges through the cathode transistor and the LED segment which briefly lights it up. The anode switches open and close at spaced intervals of 1.25uS. The LEDs are diodes so they help to isolate each inductor from discharging into each other. The discharge time is about 5uS for segments a – g and 2.5 seconds for the decimal point. The inductors have to be discharged before the cathode driver selects the next digit or those LEDs may light briefly causing a bleed effect from one digit to the next. There is not much time available for the decimal point to discharge before the following digit needs to light which explains the faster timing requirement.



If the cathode driver is faulty and the cathode switch is not turned ON for a digit, then when the anode switch opens, the inductor discharge current has nowhere to go. That current will still try to flow and most likely it will push through the junctions of the just turned off anode driver transistor. Eventually, if not straight away, the transistor will be damaged and the corresponding display segment may stay on permanently or not come on at all.



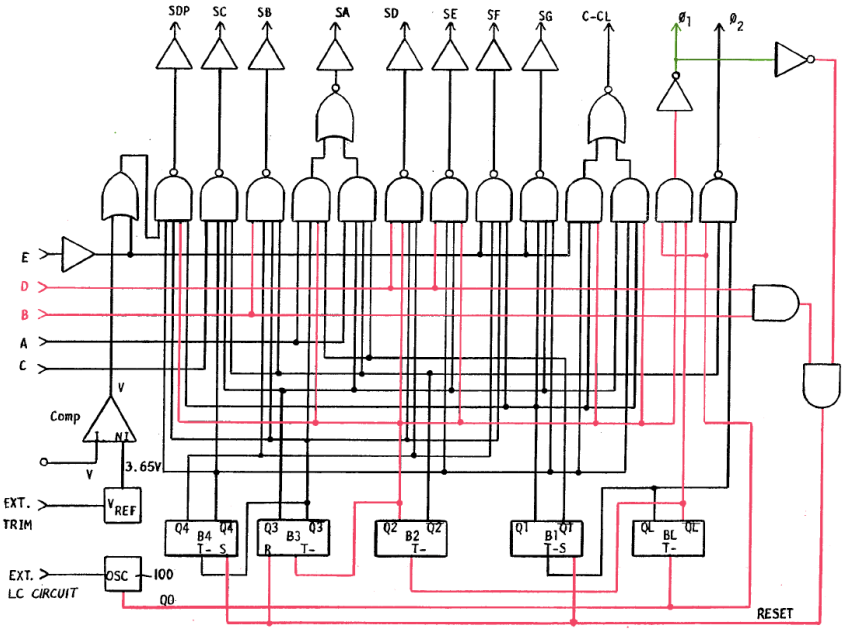
The anode driver provides the master clock pulses (Ø1 and Ø2) for all the calculator circuits, including the ARC chip. The ARC chip is responsible for partially decoding the required display information and sends it the anode driver to decode it fully for the 7 segment displays. However, the ARC chip and the anode driver don't have a complete display timing reference between each other.



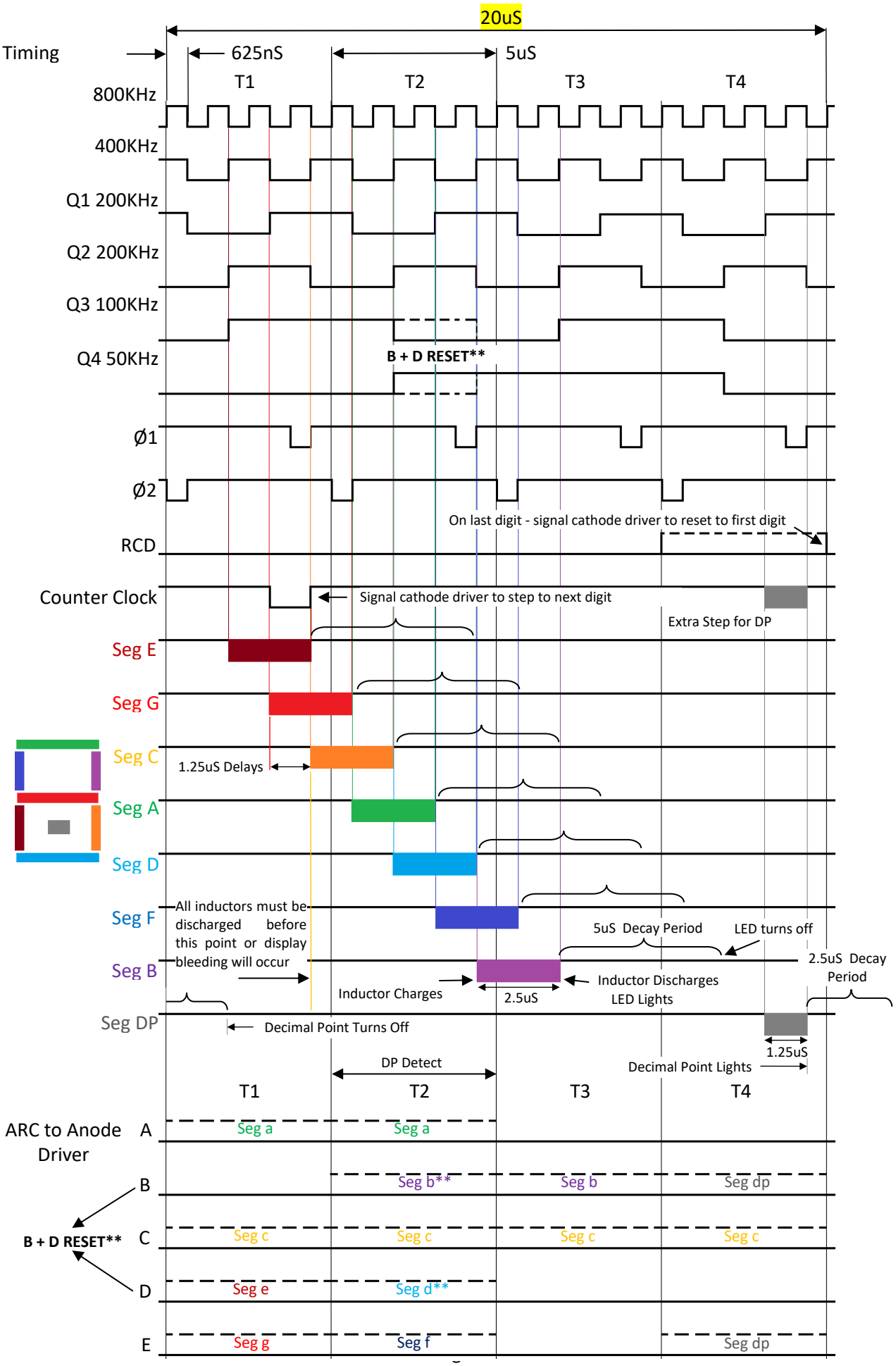
The timing diagram shows Q1 and Q2 repeat the same sequence during the T1 – T4 phases, but Q3 and Q4 have different bit patterns. These are the ones that could be out of sync with the ARC chip. If the T1 – T4 cycles are not synchronised, the anode driver cannot decode the patchy information and the display will show garbage. The method chosen to do the synchronisation is quite clever as it requires no extra data lines and is quite transparent to the display operation.

When the B and D lines from the ARC chip are logic HI and Ø1 is logic LO, the anode driver circuit detects this and sets Q3 HI and Q4 LO before the rising edge of the Ø1 pulse at T3. This then puts those clock lines in the correct logic state during the T2 cycle. The dotted lines on the timing diagram reflect this. Now if nothing changes that timing should stay synchronised until power off, but if there is a "glitch in the matrix", then the display will briefly show garbage until the timing is reset automatically during digit updates. The reset will occur on any digit that requires the B and D lines to go HI. ie. Digits 0, 2, 3, 8, 9 and 'd'. (As in HP-65 "Crd")

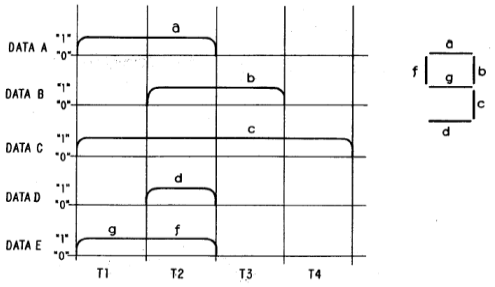
At switch on the display on the Classics always have [**0.00**] displayed so I assume then, that on the very first display refresh the digits will most likely show garbage until that first [0] is decoded, however this will happen too fast for the eye to see. The diagram shows the digital reset path when B and D go HI. Flip flops B1, B3 and B4 receive the reset pulse. Red represents Logic HI, green represents Logic LO.

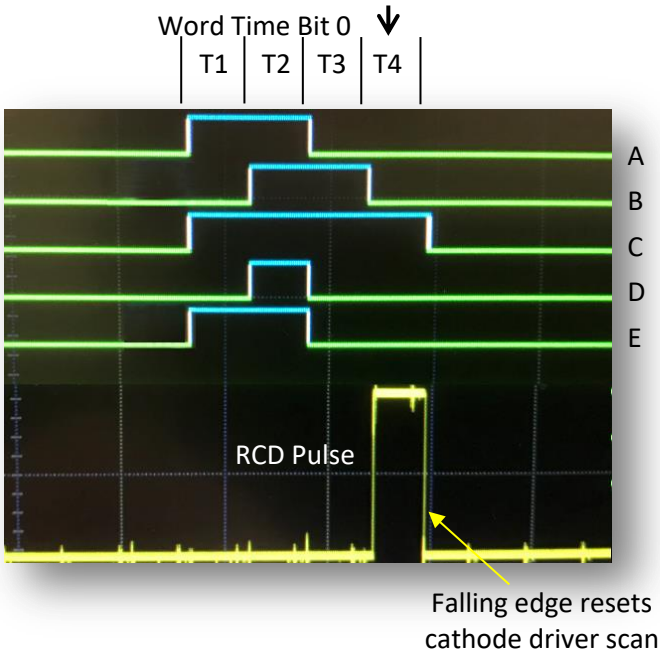


ANODE DRIVER TIMING AND DECODE



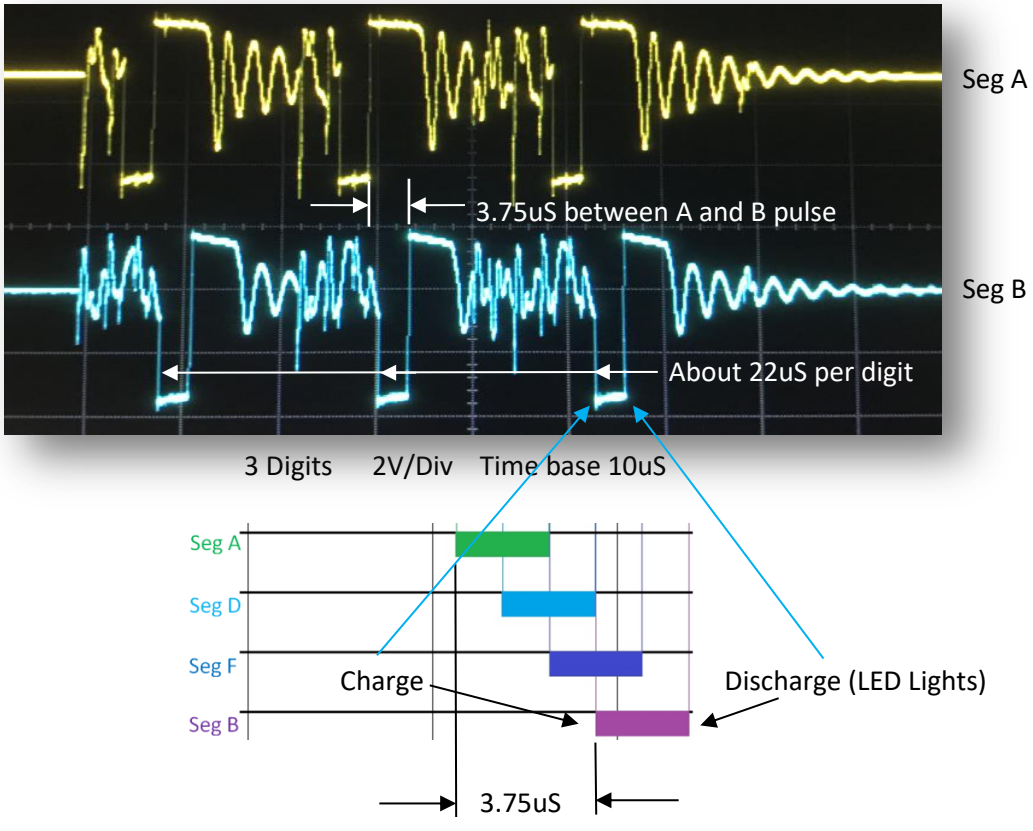
The HP-45 patent document (4,001,569) shows an image of the required data for the anode driver to decode the digit '9'. The image on the right shows the corresponding trace from an oscilloscope. The time base is 10uS. The yellow trace is set to 2V/div.



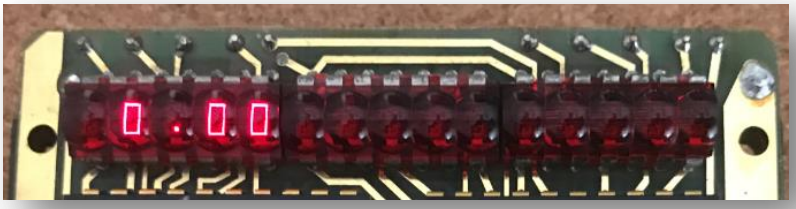


The first digit to be sent to the anode driver from the ARC is the mantissa sign, followed by the exponent ones, exponent tens, exponent sign, and then the mantissa digits 10 down to 1. The first mantissa digit (9 in this case) is the last digit to be sent to the anode driver before the cathode driver is reset to start the next display scan. This reset is caused by the falling edge of the RCD pulse.

This next trace shows the voltage appearing on the anode driver pins for LED segments A and B while the LED display is showing **0.00**. The timing discrepancy is due to the calculator RC timing which can vary due to temperature, component tolerances and aging effects. (20us designed vs 22uS actual)



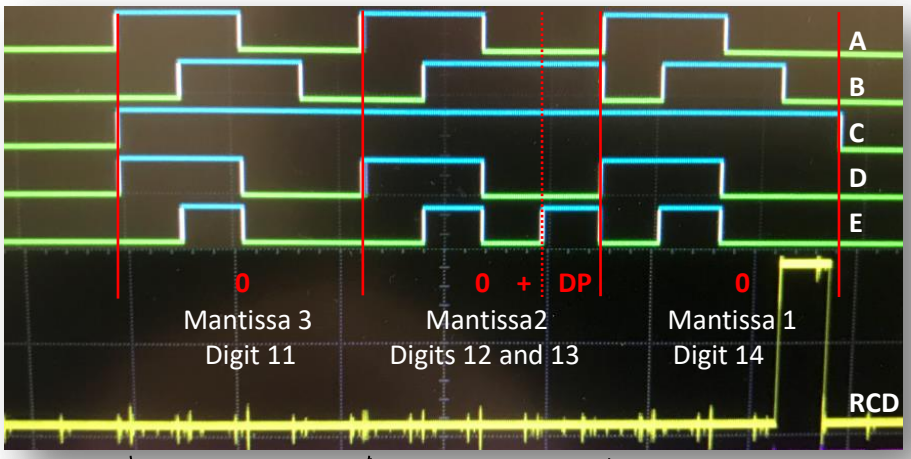
After the power is turned on, these images show a Classic display showing **0.00**, plus the trace on an oscilloscope as it monitors the A B C D and E inputs to the anode driver, and the Reset Cathode Display (RCD) line from the ARC to the cathode driver.

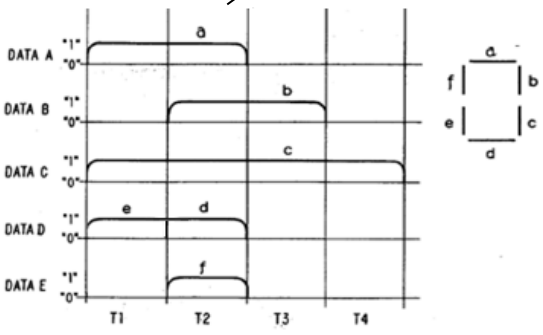


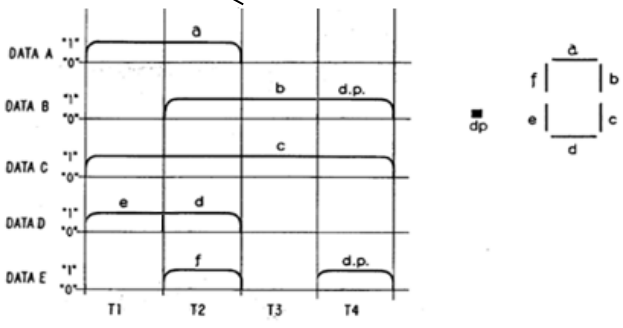
Digits 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Nibble Bits

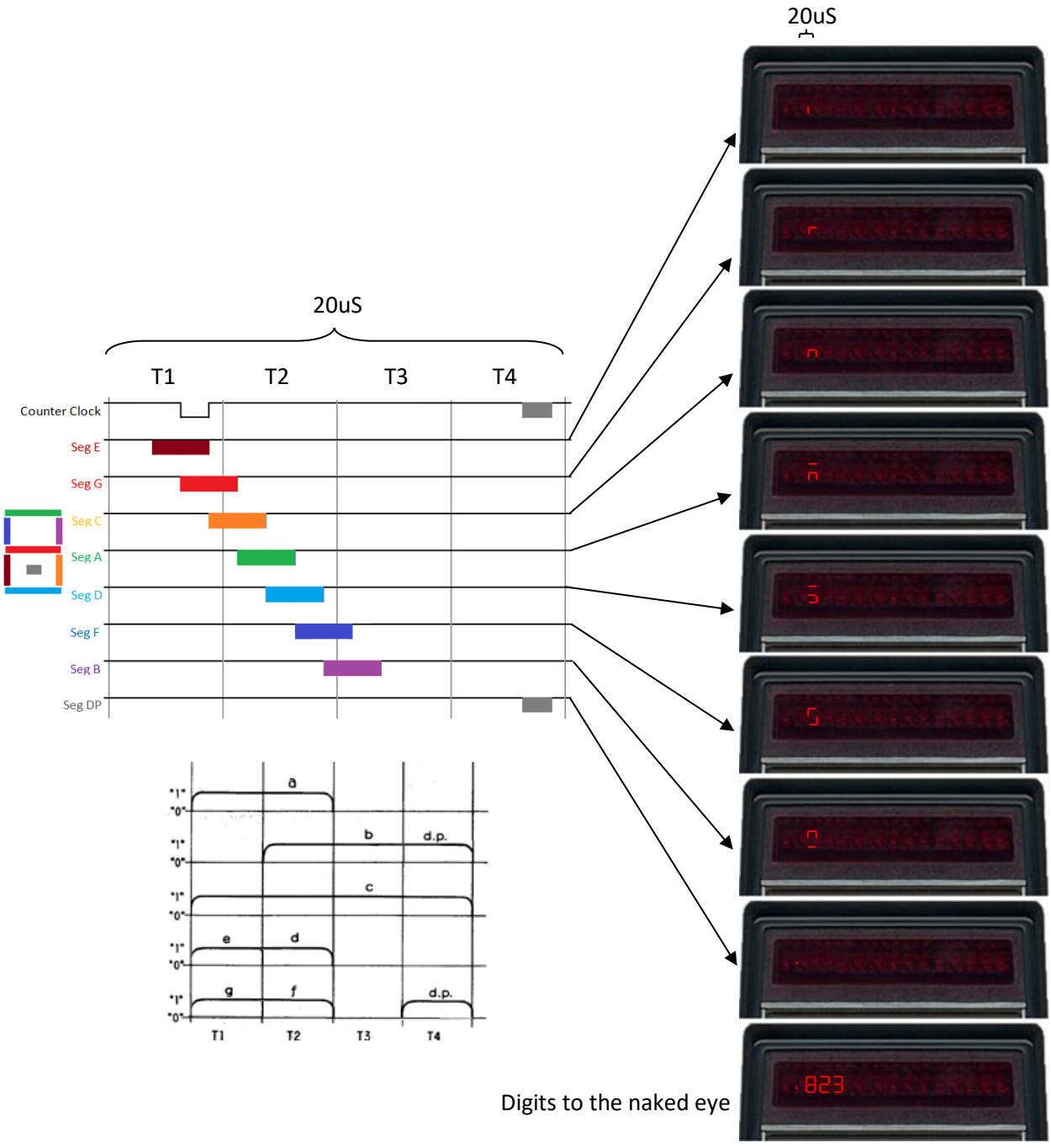
2	3	0	1	2	3	0	1	2	3	0	1	2	3	0
T3	T4	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4	T1



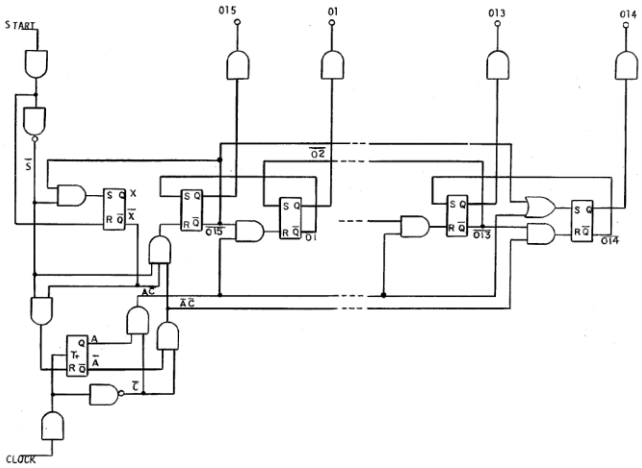




As mentioned, not all LED segments are lit at any one time, however the display refresh process is so fast that the eye only sees whatever is meant to be displayed. The following single digit (**.8**) sequence with a display showing **.823** takes about 20uS, including the decimal point. By staggering the time that each LED turns on, the current inrush that would occur if all segments were turned on together is reduced. Notice that only 1 digit is active at any one time and the LEDs in all the other digits are turned off. They will be excited in turn as the cathode driver sweeps across the display.



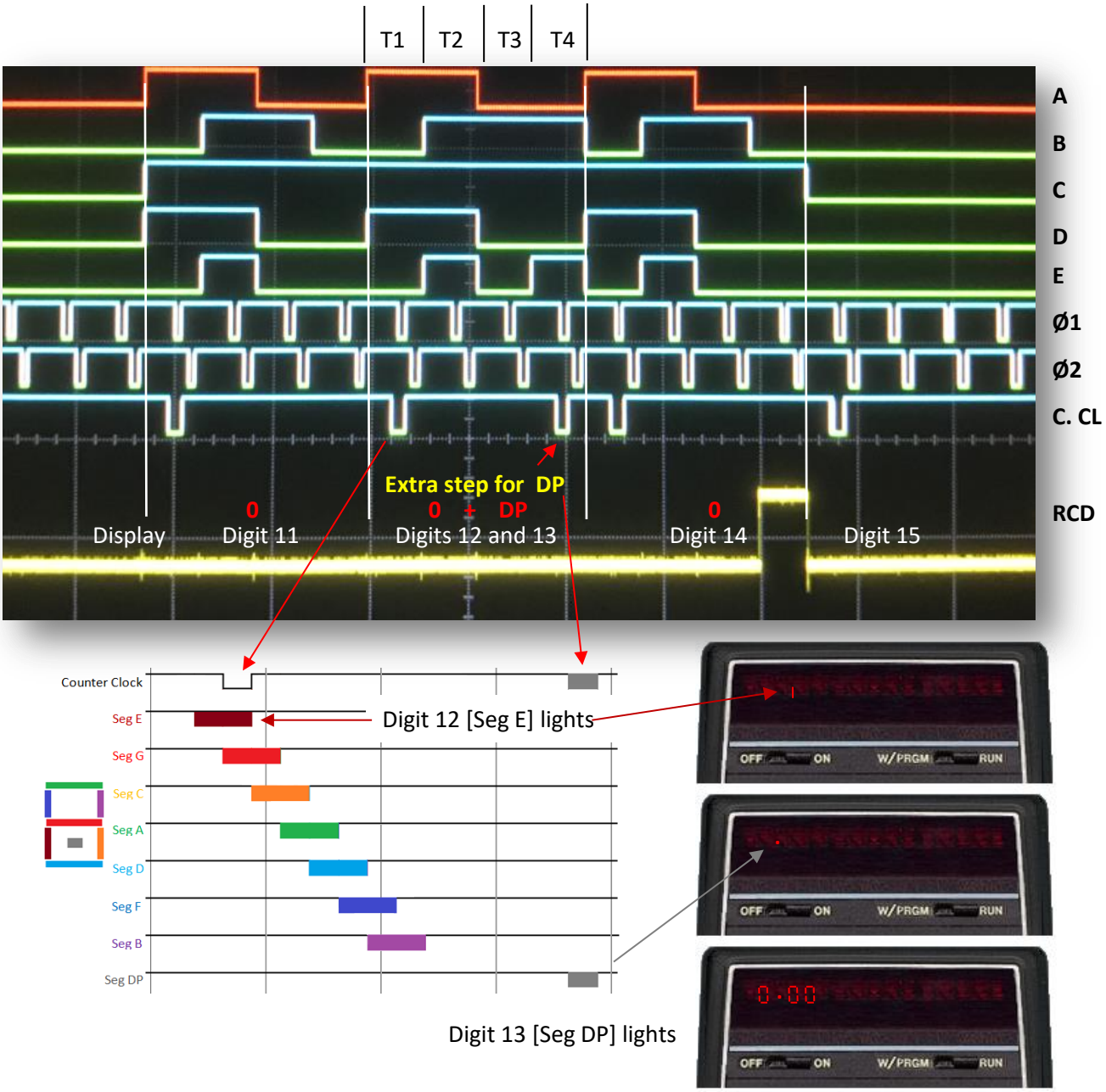
The schematic of the cathode driver chip shows that on reset, digit 15 is first out. This is the Mantissa Sign.



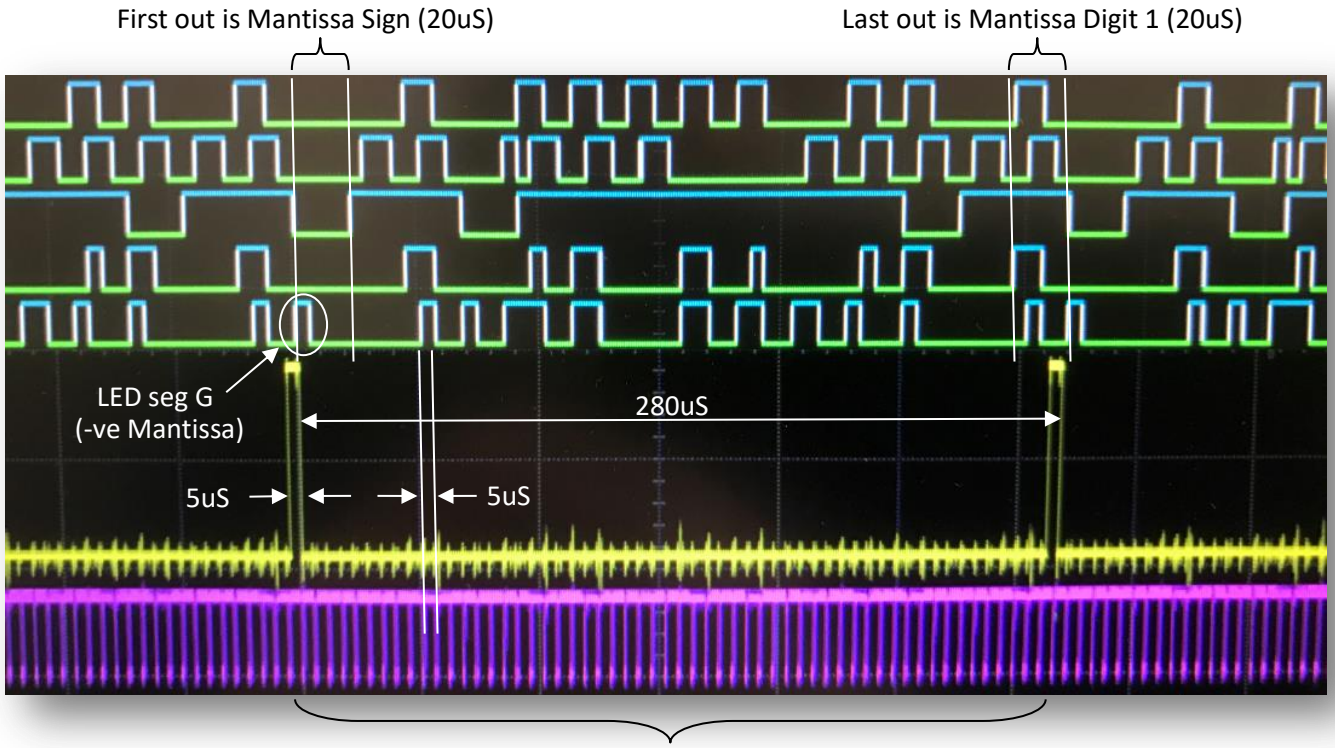
This logic trace shows the A – E pulses, Ø1, Ø2 and the anode to cathode driver step signal. (C. CL) The time base is 10uS. The yellow line is the RCD pulse - 5V/div. The display is showing **0.00**.

To get this register A and B would be set as follows.

A **00000000000000**
B **02009999999999**



This next trace shows the 56 clock pulses between each of the RCD pulses. This represents one Word (or instruction) time for the classic calculator and takes 280uS.



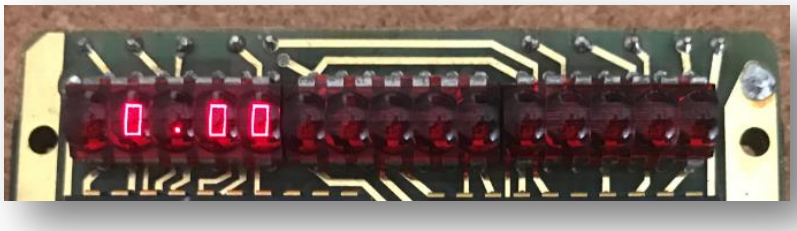
The time base is 50uS. The yellow trace is set to 2V/div, the purple trace is set to 5V/div.

See if you can decipher the digits shown on the display ☺

Hint, if you can count to 10 you are on the right track.

A B C D Com

E F H G Com



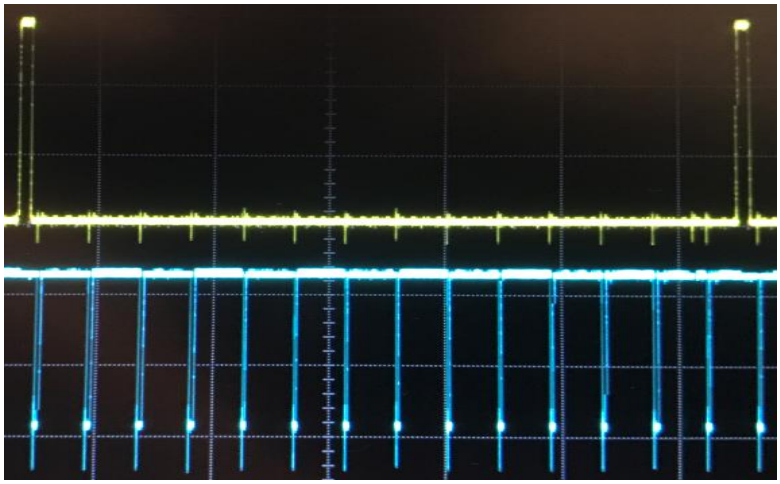
Resistance between Common and A B C D E F G is 5 ohm approx.
Resistance between Common and H (Decimal Point) is 3 ohm approx.

The Classic display driver LEDs are connected together in a multiplexed method and the Anode and Cathode drivers are responsible for sending the display information to the correct LED segment.

The Anode driver, as discussed, decodes the five display lines into 7 segment information and outputs that data to the LED anodes. The Cathode driver is responsible for selecting the correct LED digit and does this by sequentially enabling each LED digit in turn. At the start of a display sequence, the RCD line is cycled to reset the Cathode driver to start a new display sequence.

Fourteen pulses are sent to the Cathode driver from the Anode driver every display sequence. One RCD pulse is sent from the ARC on the main CPU board, for every display sequence. You might notice that the display refresh for this calculator is 310uS, not the specified 280uS. This is most likely due to changes over time with the oscillator components. The following image shows this process. The horizontal scale is 50uS/Div. The vertical scale is 2V/Div.

ARC RCD Pulses

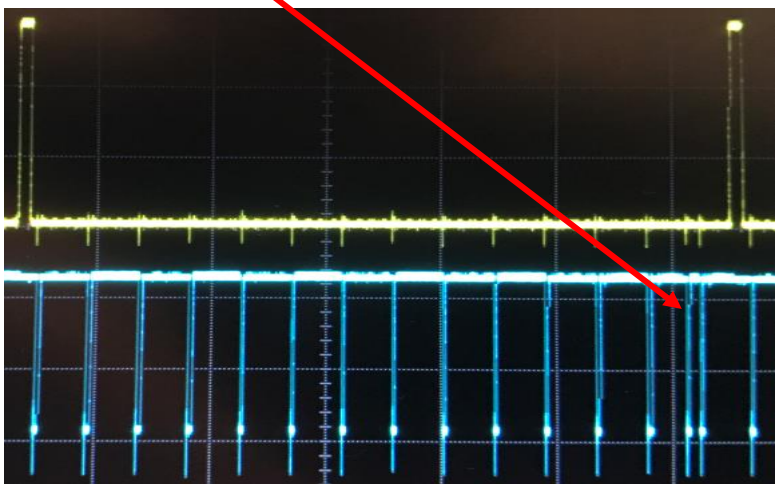


Anode Step Pulses 15 1 2 3 4 5 6 7 8 9 10 11 12 13 15

Note that in this image, there are no decimal points being shown on the display. The calculator is a HP-65 and the PRGM/RUN switch has been placed in the PRGM mode with the display showing **00 00.**

When the PRGM/RUN switch is placed in the RUN position with the display showing **0.00**, you can now see the decimal point has been decoded and the extra step has been added by the Anode driver.

ARC RCD Pulses

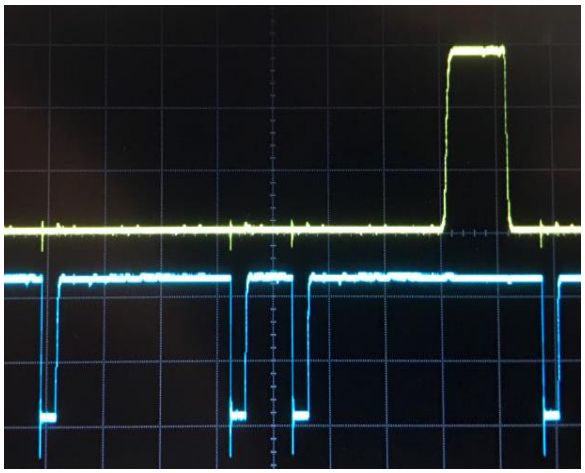


Anode Step Pulses 15 1 2 3 4 5 6 7 8 9 10 11 12 13 15

14

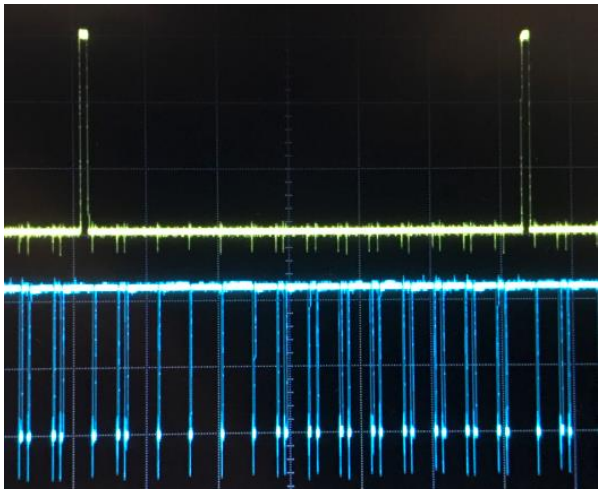
The next image is a close up of the relationship between the RCD pulse and the first step pulse.
The horizontal scale is 5uS/Div. The vertical scale is 2V/Div.

The next image shows a close up of the extra step for the decimal point.



At the hardware level, the decimal point will be included on the display when any of the Register B digits have bit 2 set to 1. That is, any digit 2, 3, 6 or 7. This is a snapshot of the HP-65 signals while running the small program listed below. While the program is running, the HP-65 internal microcode transfers much of the 2367236723 number to Register B. This scope trace shows that there are multiple decimal points lit on the display.

```
LBL A
      2367236723
      ENTER
      GTO A
```



Display Decoding

For those making emulators for the Classic series, you might be interested in how the decimal point and display decoding works. This explanation came about by chatting with Z80Sarah who was kind enough to share the experience in trying to figure this out. It may not be 100% accurate because of the lack of internal chip design knowledge but we think it is pretty close.

A Nibble 13

Nibble 0

B

Register **A** is set to display each of the digits in sequence and you can see this clearly. A **9** in nibbles [2] or [13] in the **A** register displays a negative sign. The **B** register serves as a display mask and when a nibble in the **B** register = **0**, that digit is turned on, and if any nibble = **9** or **8**, (bit 3 = 1), that digit stays off or is blanked. You can clearly see that as well. Now, when a nibble in the **B** register = **2**, the Cathode Driver is fed with an extra pulse and the decimal point is displayed in the following digit.

The values **9**, **2** or **0** for the display modifiers are used because all of the calculator processing is done with BCD numbers 0 – 9. **B** Bit 3 is tested for blanking, so B nibbles **8** or **9** will work. **B** bit 1 is tested for a decimal point, so **3**, **6** and **7** will work too. Using the values **9** and **2** might have been easier to code with. For the signs, the value **9** needs to used.

If you look back at the waveforms for displaying the digits **0.00**, the decimal point position does not seem to conform to the data in registers **A** and **B**. Similarly, the display above shows the same problem. The cathode driver actually gets an extra pulse during the time that digit **2** is being displayed, but that is nibble [11] in register **B**, not nibble [12] where the decimal point mask (**2**) is in the **B** register.

If you follow register **B** from right to left, which is the actual sequence for updating the digits, then when you get to nibble [11], there is no **2** in the **B** register for the decimal point, **B**[11] = **0**, so how can the extra increment occur for the decimal point when it hasn't been decoded yet.

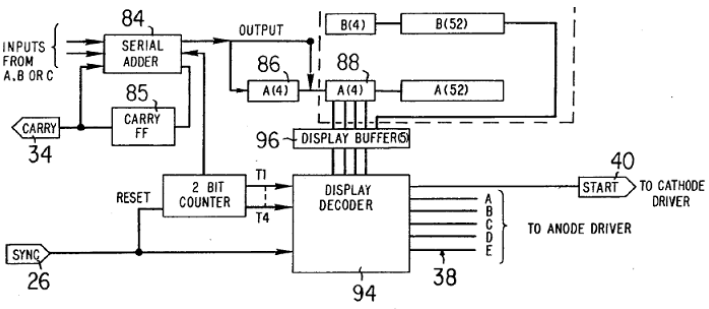
It seems like registers **A** and **B** are scanned from left to right to get the correct decimal point position. Then at nibble 12 where **A**[12] = **1** and **B**[12] = **2**, **1** is displayed then the cathode driver is stepped to the next digit, then the decimal point follows, followed by another step. So in this scenario, you would get **A**[13]**9** and **B**[13]**0** = **–**, then **A**[12]**1** and **B**[12]**2** = **1** and an extra step for the **DP**, then **A**[11]**2** and **B**[11]**0** = **2**, and so on. Decoding in this way appears to work the way it should, however both registers **A** and **B** update from right to left as shown in the oscilloscope traces. So what is going on?

The solution is because of the way the digits, negative sign, decimal point and blanking are decoded. Looking again at the decoding diagram, you can see that the data on Anode Driver lines A – E is decoded in real time and this makes sense as the circuit for the driver does not show any logic to store incoming information.

The display updating is similar to ROM instruction decoding where one instruction is being fetched while the previous fetched instruction is being executed. For the display, the display information is being fetched while the previous information is being displayed. There is one slight twist to this which is the decimal point.

Because the calculator circuit has serial data paths, the bits in registers **A** – **F** are constantly being cycled so they can be accessed by the various parts of the circuit. Bit 55 -> 54, 54 -> 53, … 0 -> 55 etc. This is shown in the HP-45 patent document, Fig. 11, as rotating arrows. These registers have to do this because just like Dynamic RAM, the bits need constant refreshing to properly store the information.

The small discussion about the C register shows that bit 0 of the registers is aligned to bit 1 of the 56 bit word. Another thing worth noting, is that T1 is aligned to word bit 1 as shown at the bottom of this page.



The circuit shown at right is part of the Arithmetic & Register Circuit (ARC) IC. The block marked (88) is the 4 least significant bits (LSB) of the 56 bits from register **A** which, as mentioned, is constantly cycling during normal operations. The Display Buffer (96) is used to isolate register **A** (88) and the Display Decoder (94) during T1 – T3. It can also modify the 4 bits as the Display Decoder receives them in T4. There is also a single bit path from register **B** to the Display Buffer which is always the current bit 0. The Display Decoder has a connection to a 2 bit counter that provides the references for the T1 – T4, and although not shown, there is a connection to a display On/Off flip flop controlled by the instructions `Display Off` and `Display Toggle`. The change from Lo to Hi on the SYNC line on word b45, makes sure the 2 bit counter is reset.

During T1 – T3 the partially decoded display digit data is continuously being sent via the A – E lines from the Display Decoder to the Anode Driver which fully decodes it in order to drive a single LED digit in the display as shown in the Decoder diagram. As mentioned, while register **B** is shifting, the current LSB is presented to the Display Buffer. At time T1 this is bit 0 of the current **B** nibble, T2 = bit 1, T3 = bit 2, and T4 is bit 3.

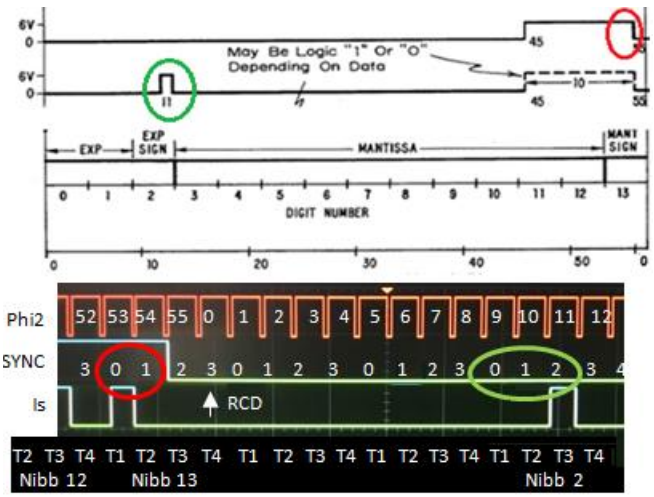
During time T2, bit 1 of a **B** nibble is at the display buffer. This bit has no use now but is required during T4 as explained later. However, it will be overwritten during the next T3 and T4 so a flip flop is set to remember this bit state. During time T3, **A** bit 2 is shifting into the Display Buffer, and **B** bit 2 is not required.

T4 is where a lot of things happen, although the Anode Driver is essentially idle as all LED segments are now off. The Display Decoder does not change the state of A – E during T4, they stay the same as they were in T3. This allows the **A** data to enter the Display Buffer and connect with the Display Decoder without affecting the A – E outputs.

Block 88 now holds a 4 bit nibble from register **A** and this is transferred to the Display Buffer but it may be modified during the transfer. **B** bit 3 is now at the Display Buffer and it controls digit blanking. Remember the **9** in the **B** nibble for a blank digit.

This bit is logically OR'd with the display On/Off flip flop output. The result is then OR'd with each of the bits being transferred to the Display Buffer. So if the OR result = 1, the buffer data is automatically changed to 1111 which decodes as a blank digit, otherwise the buffer data stays as is. (Logically OR'ing bits with 0 has no effect).

The patent document mentions that the Display Decoder tests the **Is** bus (Green) and the SYNC line (Red) to let it know that the current digit is either an Exponent or a Mantissa sign. If this is the case *and* the nibble in the Display Buffer = **9**, the nibble will be changed to show a sign digit **-** instead of **9**.



However, the SYNC pulse (red) ends at Word bit 54 and at this time the buffer only holds 2 bits from **A** (0 and 1) The **Is** pulse (green) at word bit 11 aligns with **A** bit 2 so there are only 3 bits in the nibble (0 1 and 2) at this time. So at both times, the Display Decoder has no way of knowing yet if the **A** value will be **9**.

To make this work, the falling edge of the SYNC line sets a sign flip flop in the decoder, while the falling edge of the **Is** pulse at word b11 does the same. When the **A** nibble data is made available to the Display Decoder at T4, a test can now be made to see if the nibble is **9**. If it is *and* the sign flip flop is set, then the nibble value is changed and decoded to show a negative sign. The change probably sets bit 1 or bit 2 to 1 making the sign code 1101 or 1011. The sign flip flop is reset during T1. If a sign in **A** and a blank in **B** appeared at the same time, the blank would win as all bits would be 1 from the OR gates.

Yet another thing happens during T4, and that is the decimal point processing. Remember the nibble value **2** in register B? In binary this has bit 1 set. The flip flop that remembered **B**[1] during T2 is now tested and if it is not set, nothing happens. If the flip flop is set, then the Display Decoder immediately sets the B and E lines High. The Anode Driver logic will see this and because it is time T4, will step the Cathode Driver to the next digit and then turn on the decimal point LED. It can do this without upsetting the display because as you may recall, the Anode Driver was just sitting idle at this part of T4.

This is quite complex but is why the decimal point appears to be lit before a **B 2** is decoded. The decimal point LED will be turned off during the next T1, and the Cathode Driver will receive a step pulse from the Anode Driver allowing the next LED digit to be displayed.

One last thing happens during T4 if nibble 13 is being processed. The Display Decoder resets the Cathode Driver ready to start a new display refresh. This means that the last digit out of the Display Decoder, which is normally digit 15 on the display (the Mantissa Sign), is actually controlled from the Cathode Driver's first output, not the last. This is because of the *process while execute* pipeline of the display logic. Digit 15 is the first out, shown on the Cathode Driver drawing, and also on the scope trace.

It is possible as shown previously, that multiple nibbles in the **B** register can have bit 1 set which will show multiple decimal points, however due to its internal logic, once the Cathode Driver has stepped to digit 14, (remember digit 15 is the first), it cannot step any further.

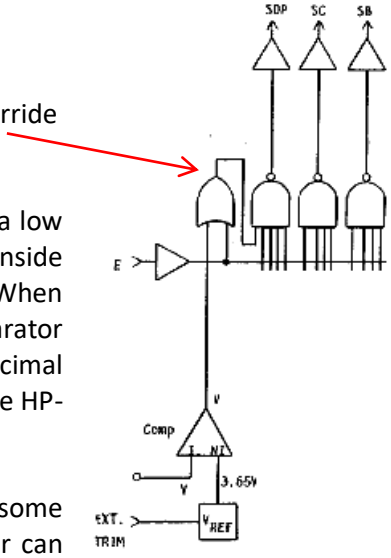
With this logical arrangement, decimal points can never be shown in 2 adjacent digits and the decimal point cannot appear in the Mantissa Sign because that would require a **2** in **B** nibble [14] which does not exist.

Example: **-123..456** would show as **-123.4.56**
.123.456 would show as **123.456**

It is possible for the signs to show numerical digits.

A 01000000000999 would show as **1.00.**
B 02029999999999

DP Override



You might then ask, how can the Classics light all the decimal points for a low power condition? This function is controlled by a voltage comparator inside the Anode Driver which tests the battery voltage against a reference. When the battery voltage is less than the reference, the output of the comparator goes HI and overrides the normal decimal point logic. This forces the decimal point to be on for all digits as shown in this partial circuit from Fig 17 of the HP-45 patent.

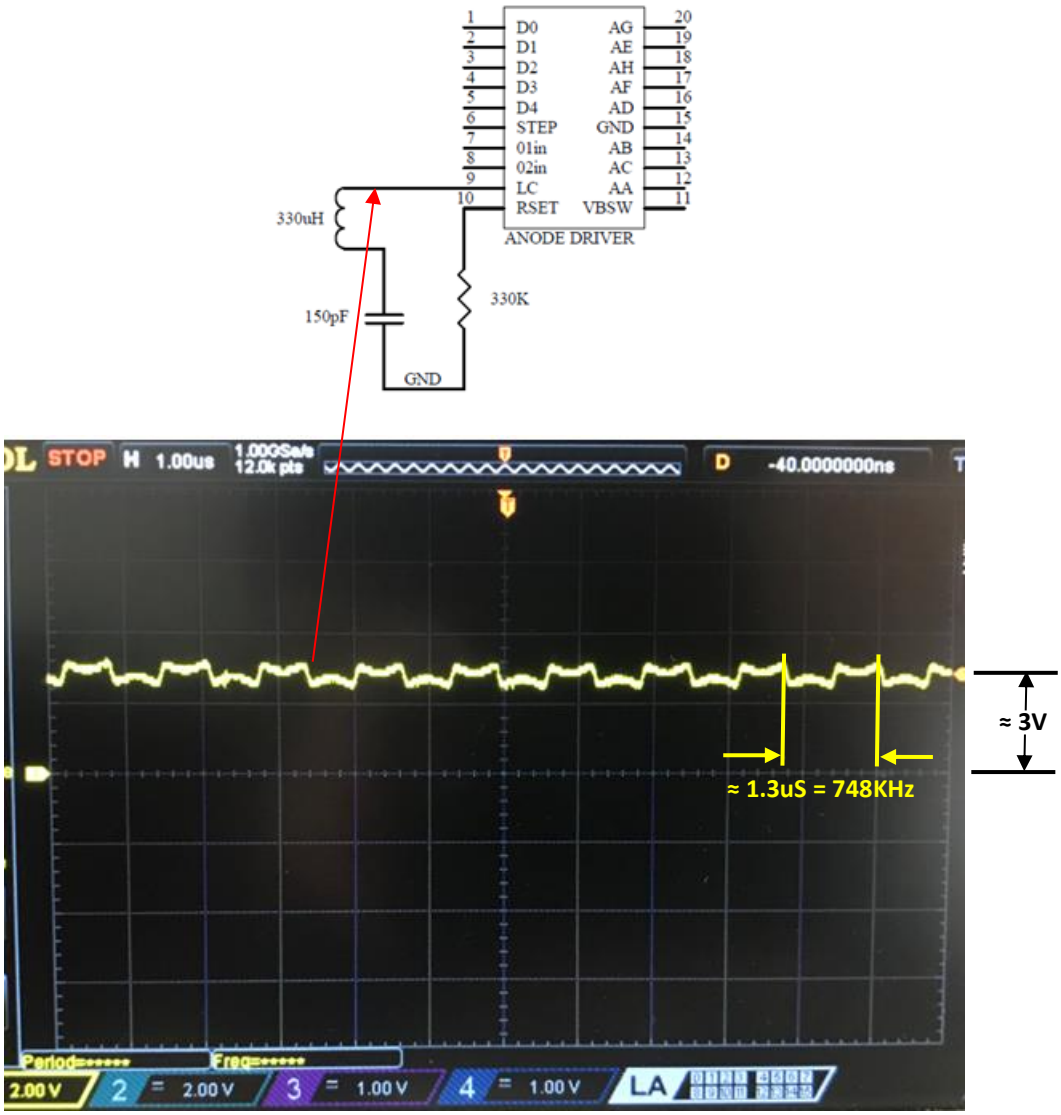
All in all, the Classic LED display is quite a complicated setup and requires some intricate logic and timing to run properly. A modern day microcontroller can format data for a Classic LED display quite easily, but then, hey, those weren't available back then.

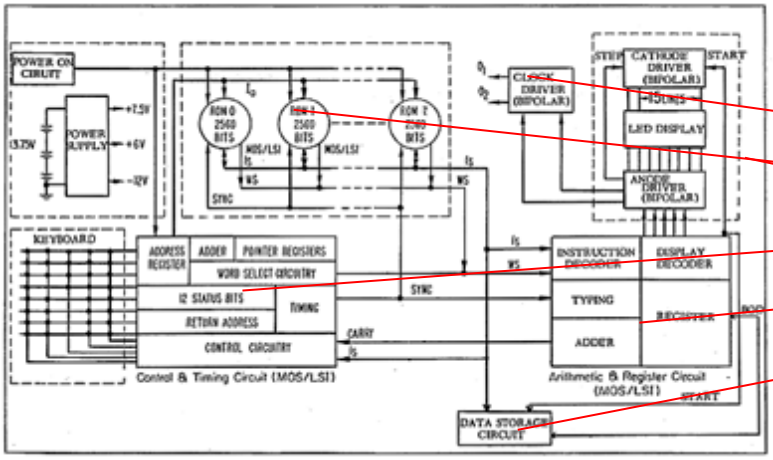
A processor algorithm might be as follows where the display buffer is output to a multiplexed display starting at the mantissa sign and ending at the exponent unit.

```
Define 8 bit LED patterns for Blank, Decimal Point, negative, and digits 0 - 9
Create a 15 digit display data buffer
buffIdx = 14;

for I = register_nibble_13 down to register_nibble_0 do
       if B[I] and 8 <> 0 then
         Buffer[BuffIdx] = LED_Blank // digit is blanked
       else
         if B[I] = 2
           Buffer[BuffIdx] = LED_decimal_point
           BuffIdx = BuffIdx - 1 // shift to next digit
           if BuffIdx < 0 then
             Stop
         if (I = 2) or (I = 13) then
           if A[I] = 9 then
             Buffer[BuffIdx] = LED_Negative
           else
             Buffer[BuffIdx] = LED_Digit_Array[A[I]] // 0 - 9
       BuffIdx = BuffIdx - 1
       if BuffIdx < 0 then
              Stop
```

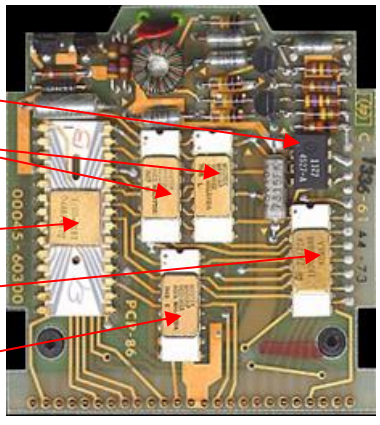
The Teenix calculator emulator has a Classic display logic simulator screen which may help to understand how this explanation of the process works.

HP-55 Oscillator waveform captured on the crystal connection to the anode driver





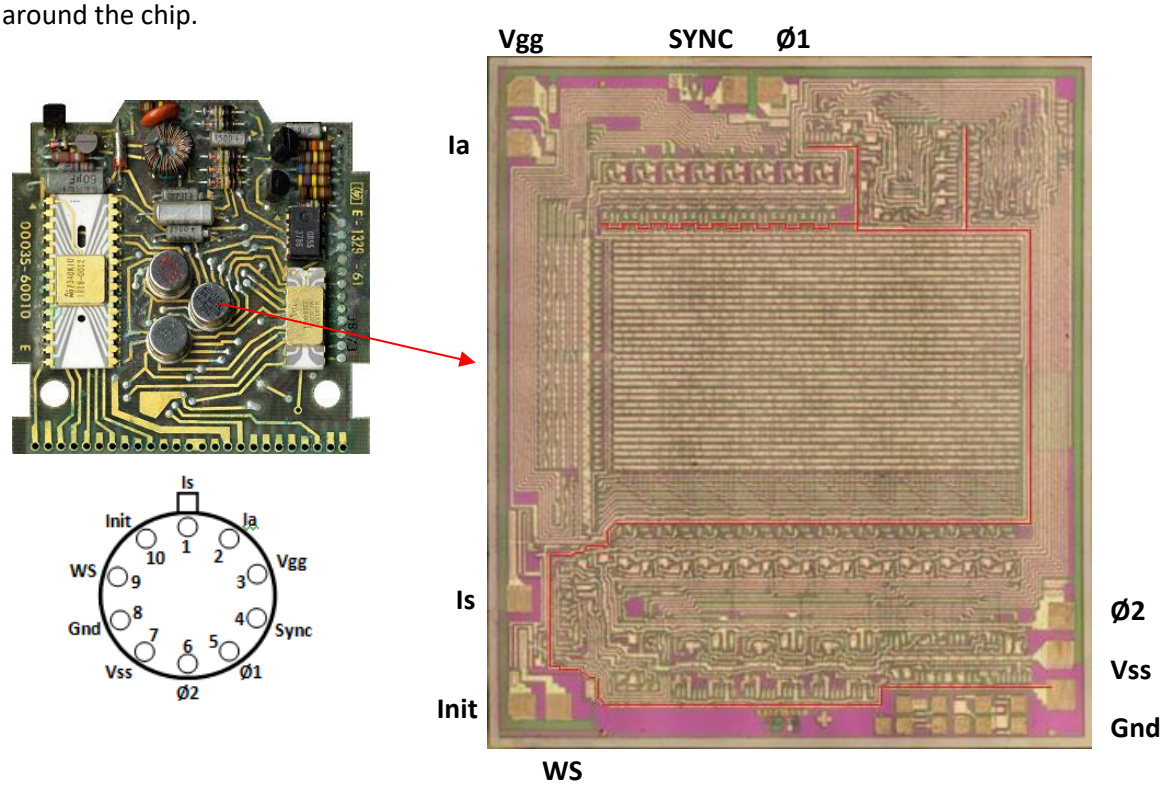
Block Diagram of the Classic Series



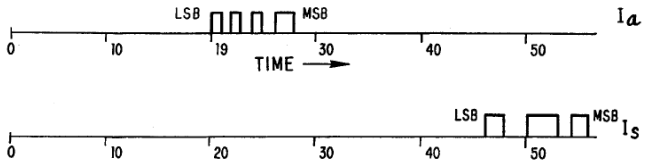
HP-45

There are 8 ROMs available in the HP-45, and each is selected by the instruction `select rom (x)` where (x) can only be a value between 0 and 7. On initial power up, ROM 0 is selected by default. That way the calculator will always start executing code from ROM 0, address 0. If the microcode needs to begin executing in a different ROM then appropriate `select rom` instructions will be executed. There are 256 instructions stored in each ROM, and that requires an 8 bit address be used to access them all. Each instruction is 10 bits wide. The ROM chips used in the HP-45 are called quad ROMs meaning that 4 ROMs are packaged in one chip.

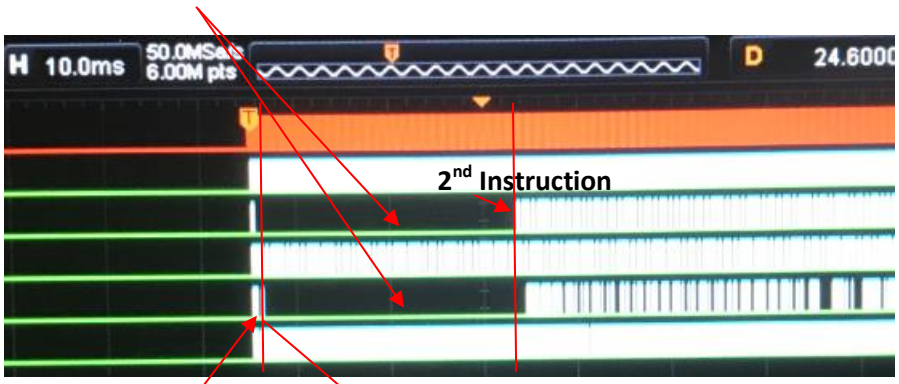
Shown below is a nice image of the HP-35 ROM chip taken by Peter Monta. The red highlight shows the Gnd trace around the chip.



Each instruction requires 56 clock cycles to execute. This image taken from the HP-45 patent document shows the IA and the IS buses. The IA bus is used to transfer the instruction address which is used by the ROMs to get the next instruction. This instruction is then sent out from the active ROM on the IS bus. After the address is accepted by the ROM, it takes a few clock cycles to process the address and fetch the instruction before it can be sent out. You can see this delay in the diagram.



This image shows the various buses at switch on. You can see below that the IA line is low for most of the initialization. During this time the ROM address is continually set at 0 during clock pulses 19 to 26. The WS line is also low because there is no WS information encoded in JSB instructions.



The first 10 bit instruction is accessed here

LSB 1 0 1 1 0 1 0 1 0 0 MSB

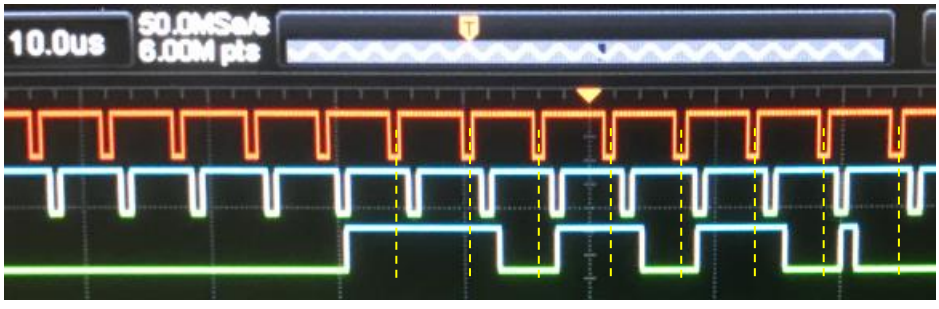
Here is first line of microcode shown in the HP-45 patent document. The dots represent 0's.

```
0    L00000:   ..1.1.11.1   ->   L0053     PW01    JSB PW02
```

Looking at the instruction shown in the image we get 0010 (2) 1101 (B) 01 (JSB), in other words JSB $2B. As you can see the first instruction is indeed a JSB. During initialization, this single instruction executes continually. After initialization has completed the IA bus becomes active and the next address on this bus will be 2B hex which came from the JSB instruction.

```
ROM ADDR      CODE
$00 00        pwo1:  jsb pwo23
$00 2B        pwo2:  go to pwo3
```

The image below starts from the 2nd instruction (marked above) and shows the next address on the IA bus is 2B hex.

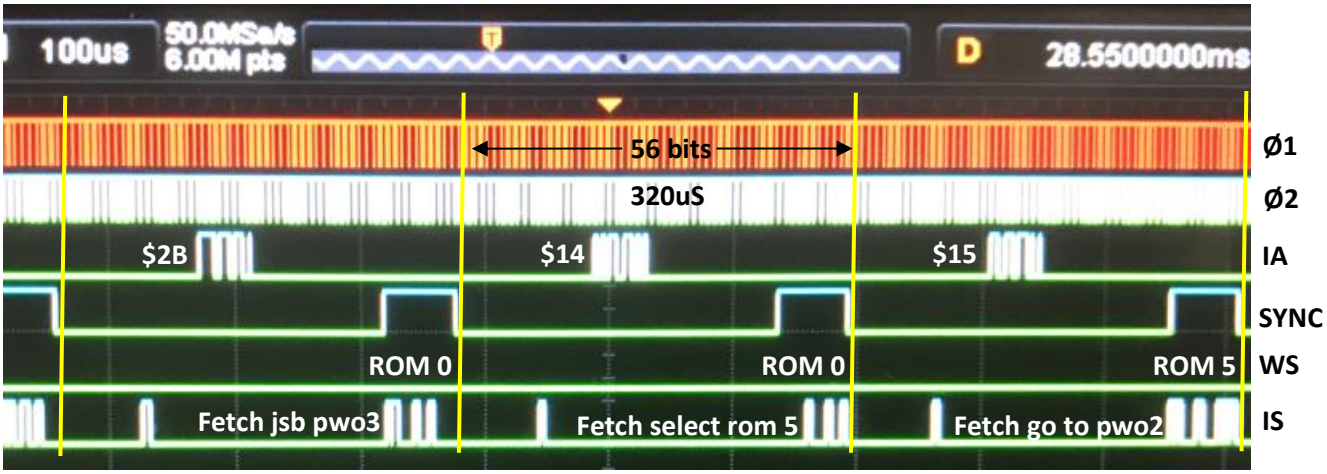


LSB 1 1 0 1 0 1 0 0 MSB
B 2

Next we see the first four instruction execution sequences. Instruction 0 (`jsb pwo2`) has already executed. The yellow bars indicate the 56 clock cycles for each instruction.

```
ROM ADDR      CODE
$0000         pwo1:    jsb pwo2
$002B         pwo2:    go to pwo3
$0014         pwo3:    select rom 5
$0515         pwo2z0:  go to pwo2
```

The master clock is 784KHz divided by 4 which gives 196KHz. Divided by 56 bits per instruction, this is about 286uS per instruction, or about 3500 instructions per second. You can see that this particular calculator is executing instructions at around 320uS as shown by the 100uS scale graticule.



Execute jsb PW02 Execute jsb pwo3 Execute select rom 5

Notice that the Program Counter shown on the IA bus has incremented normally from $14 to $15 after the `select rom 5` instruction, but now the code is being fetched from that address in ROM 5.

You can also see that an address is set and then the instruction there is fetched from ROM during a 56 bit cycle, but that instruction is not executed until the next 56 bit cycle. In that way, one instruction is being executed while the next instruction is being fetched.

The logic analyser image below is showing the instruction flow for a HP-45 in the code loop that waits for a key press. This code is being executed from ROM 3 which would have been selected from the following instruction which is located in ROM 6 at address $FE.

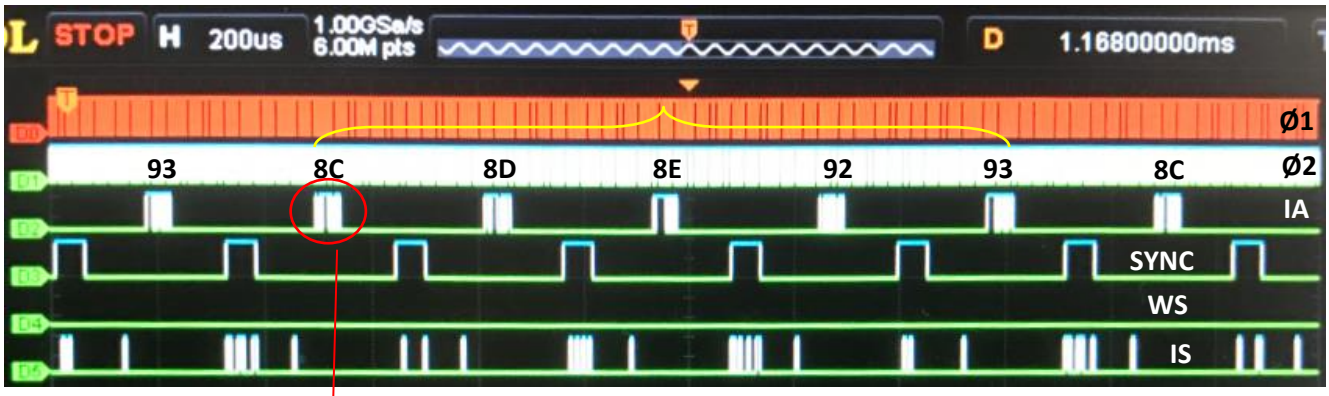
```
ROM ADDR      CODE
$06FE         ret3:    select rom 3
```

This is the next instruction that executes which is a return from subroutine..

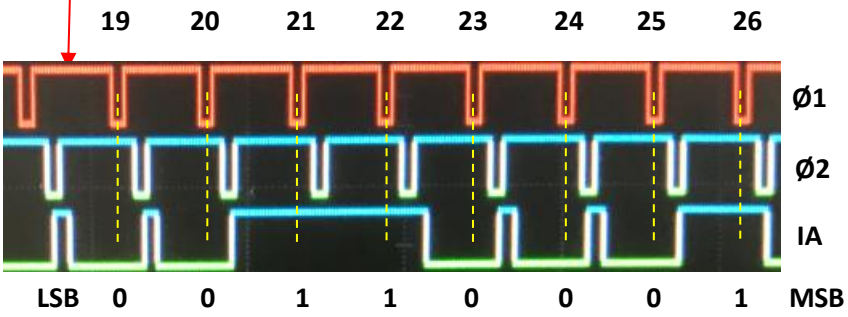
```
ROM ADDR      CODE
$03FF         retnzx:  return
```

Some other instructions will be executed before finally ending up in this code loop which continually scans to see if a key has been pressed.

```
ROM ADDR      CODE
$038C         dsp3:    1 -> s8              }
$038D                  if s5 # 1            }
$038E                      then go to dsp5  }
$0392         dsp5:    if s0 # 1            }   s0 = 1 if a key is pressed
$0393                      then go to dsp3  }
$038C         dsp3:    1 -> s8
$038D                  if s5 # 1
$038E                      then go to dsp5
$0392         dsp5:    if s0 # 1
$0393                      then go to dsp3
```



The ROM address is presented on the IA bus during the eight Ø1 and Ø2 clock cycles between 19 and 26.



The address data is clocked from the Ø1 signal and the IA data becomes `10001100` which is `8C` in hex.

Notice the IA bus only uses 8 bit addresses to access each of the 256 memory locations in the selected ROM.

The data on the WS bus depends on what part of a register needs to be acted on when those particular instructions are being executed. The WS information is embedded into the 10 bit instruction in bits 2, 3, and 4.

The storage format of each 56 bit register is shown below. In this case the C register is shown.

Mantissa Sign Mantissa Exp Sign Exponent

C register `9 1234567890 912` (Decimal = -1.234567890 E-12)

The digits are Mantissa Sign (13) down to Exponent Units (0).

The P register is 4 bits wide and can hold a number from 0 to 15. It is used as a pointer to a single digit, or multiple digits, within a register. In microcode, it is also used as a delay counter. The maximum number that the P register holds depends on the calculator model. For the HP-45, the P register maximum value can be 15. As there are 14 digits in each register, numbered 0 to 13, when the P register is used as a digit pointer, its value will be limited to between 0 and 13. Here, it is set to 4. This can be accomplished by the `4 -> P` instruction.

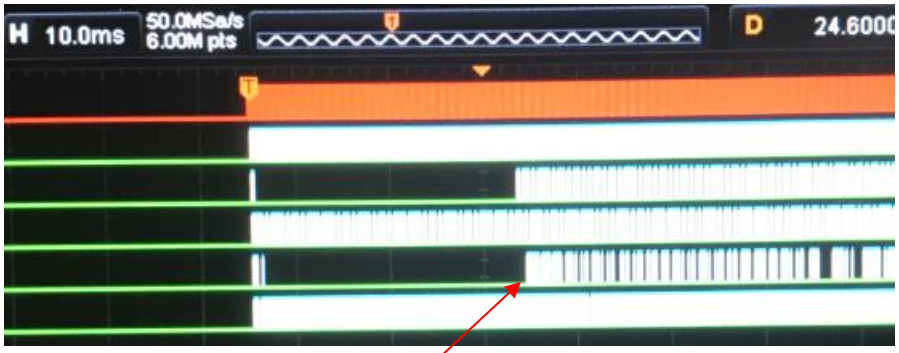
P register `4`

We will use a “clear register c” type of instruction here. This is written as `0 -> c`. The WS information is written into instructions that use it by enclosing the identifier inside brackets.

Instruction `0 -> c[?]` (? represents one of the WS types)

Using the value above, clearing the C register using the different WS methods available results in these values.

Type	Value	Use	Instruction	10 bit Binary	New C Register value
p	000	P digit	0->c[p]	00110**000**10	9 123456780 912
ms	001	Mantissa	0->c[ms]	00110**001**10	9 **0000000000** 912
x	010	eXponent	0->c[x]	00110**010**10	9 1234567890 9**00**
w	011	Word	0->c[w]	00110**011**10	**0 0000000000 000**
wp	100	Digit 0 up to P	0->c[wp]	00110**100**10	9 123456780**0 000**
ms	101	Mantissa + Sign	0->c[ms]	00110**101**10	**0 0000000000** 912
xs	110	eXponent Sign	0->c[xs]	00110**110**10	9 1234567890 **0**12
s	111	Sign	0->c[s]	00110**111**10	**0** 1234567890 912

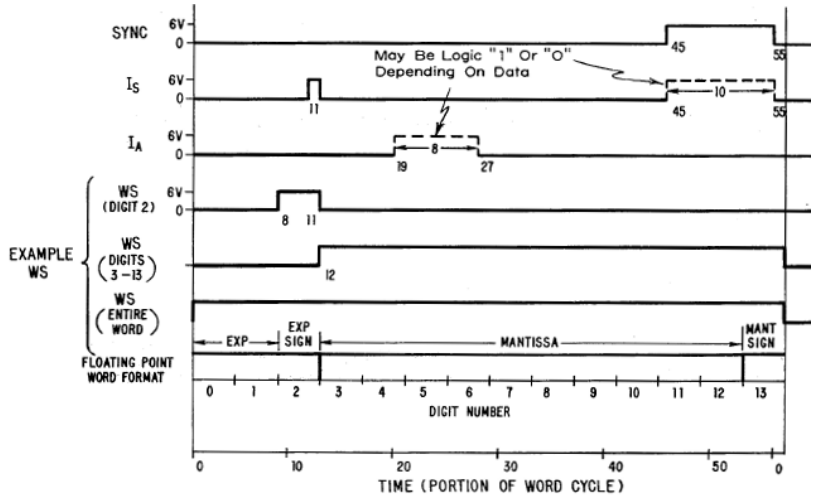


The fifth instruction that executes from switch on starts to make use of the WS bus. This part of the microcode is preparing to clear to HP-45 memory to make sure the 10 available storage registers are all set to zero.

```
ROM ADDR      CODE
$0000         pwo1:    jsb pwo2
$002B         pwo2:    go to pwo3
$0014         pwo3:    select rom 5
$0515         pwo2z0:  go to pwo2
$05E4         pwo2:    0 -> c[w]          (5th Instruction – 1st to make use of the WS bus)
$05E5                  c - 1 -> c[s]      (6th Instruction – 2nd to make use of the WS bus)
```

There are 56 clock cycles for each instruction.
There are 14 digits in each register.
Therefore, there are (56 / 14 = 4) clock cycles used to process each digit.

The WS bus will cycle 4 times for each digit that is included in the [WS] part of the instruction and will stay logic low for those digits that are not. This image, taken from the HP-45 patent document, shows how the WS bus interacts with the 56 clock cycles.



Fetch `0-> c[w]` Execute `0 -> c[w]` Fetch `c - 1 -> c[s]` `Execute c - 1 -> c[s]`

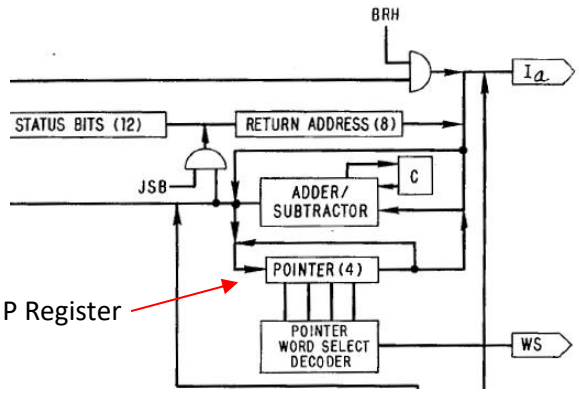
You can see while `0 -> c[w]` is executing the WS bus is cycling 4 times for each of the 14 register digits and while `0 - 1 -> c[s]` is executing only the last 4 cycles are active for the mantissa sign digit.

Magnified section from above showing the 4 WS cycles

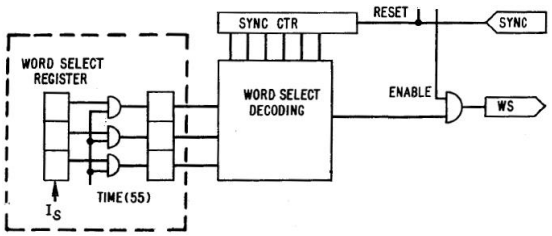
Mantissa Sign – Digit 13

The P register is located inside the Arithmetic and Control chip (ARC). As it has a role to play in the word select function, it is responsible for providing the signals for the WS bus when the P register is used to determine which digits are to be acted on.

The other types of word select functions are controlled from an unlikely place. The WS bus signals for these come from circuitry from inside the ROM chips. It makes sense to let the ROM chips produce the WS signals because they are where the instructions are fetched from. As the instructions are read from ROM memory they are checked for WS information and if so, the ROM chip will output the required information onto the WS bus. In the logic circuit diagrams below you can see that the WS bus signals exit from each of these chips.



ARC Chip



One of the ROM chips

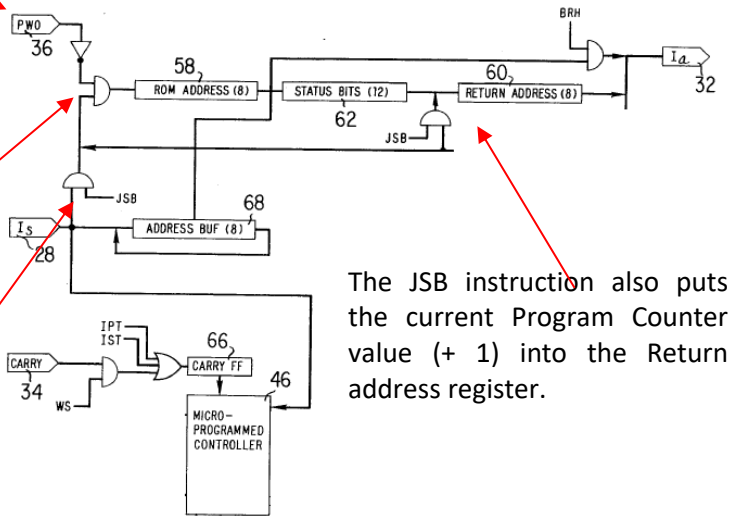
The code shown below is part of the clearing code for the ten memory registers and executes just after switch on. You can see that the WS bus activity coincides with the instruction flow.


```
$05EE          a exchange c[w]
$05EF          c -> stack
$05F0          c -> data
$05F1          a exchange c[w]
$05F2          c + 1 -> c[p]
$05F3          c + 1 -> c[p]
$05F4          if no carry go to clr3
$05EC  clr3:   c - 1 -> c[p]
$05ED          c -> data address
$05EE          a exchange c[w]
$05EF          c -> stack
```

This code repeats until the ten registers are cleared.

The ROM address register (58) register would have a random value in it at switch on as it does not appear to have a reset to zero circuit. The ROM address is inhibited from getting onto the ROM address bus (Ia) while the system is powering up. This means that zeros initially appear on Ia making sure that the instruction at ROM address $00 is the first one fetched. Also, only ROM 0 is set up to reset to the active ROM on power up.

The HP-45 patent document states that the first instruction that executes must be a JSB (Jump to Subroutine) instruction so that the ROM address register (58) is initialised properly. You can see that the JSB and the power on signal (PWO) are connected to the ROM address register though some logic gates.



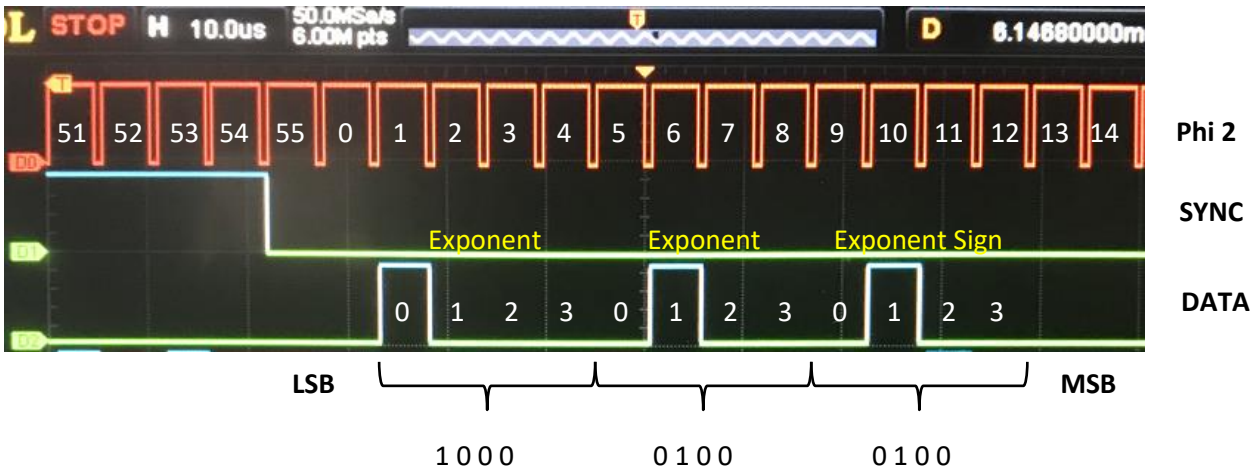
The JSB instruction also puts the current Program Counter value (+ 1) into the Return address register.

The lower 8 bits of the JSB instruction are fed into the ROM address register when it executes and sets it to a valid address. If this didn't occur as the first instruction, the second instruction that is fetched will be from a random ROM address. From here on, the ROM address register will always have a valid address in it. When a return instruction is executed, the return address register is shifted into the ROM address register, which in this case will be ROM address $01.

You may have noticed that the ROM Address, Status and Return Address registers are connected in series. This is because they all share a common 28 bit register. ROM Address = 8 bits, Status = 12 bits, Return Address = 8 bits. The data in this register is constantly circulating allowing data in it to be modified and shifted out to the correct location when required.

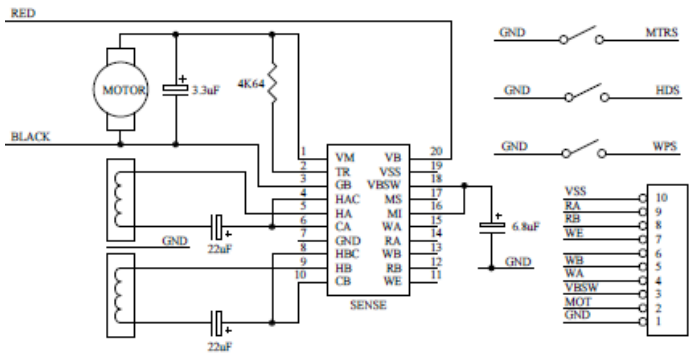
The Address Buf register (68) is used to store the GOTO address from the lower 8 bits of the GOTO instruction and is transferred to the Ia bus when the GOTO instruction executes. This instruction can only execute when the Carry Flag is clear.

This next scope image shows the output from the C register. You can see that C bit 0 aligns with Word bit 1.



The scope image was taken using a HP-65 while it was waiting for a key press. The C register and M register are constantly swapping values and at this time the C register had **00000000000221** in it.

HP-65 card reader circuit board



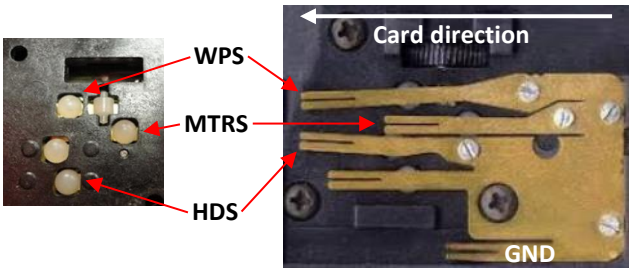
The card reader relies on mechanical switches for its operation. When the card is inserted into the slot, the first switch to close is the motor switch (MTRS). This alerts the Card Reader Controller (CRC) chip that a card is present and to start reading or writing. If the PRGM/RUN switch is in the PRGM position, the CRC will start a write process, or if the switch is in the RUN position, it will start a read process.

The switches are activated by the card rubbing over small nylon balls which rest against spring contacts. The card presses down on the ball forcing the contact to connect to the circuit board pads beneath them. This effectively shorts the contact to GND.

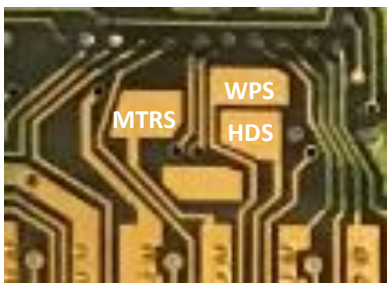
There are three switches listed in order of operation.

MTRS	Motor Switch	Alerts the CRC that a card is present in the slot and to start the motor.
HDS	Head Switch	Alerts the CRC that the card is sufficiently in the slot to decide what process to begin, either read or write.
WPS		Alerts the CRC of the magnetic card is write protect status.

A new magnetic card has square ends on it. If the user wants to write protect the card, then the corner that is inserted first can have one corner cut as shown below. Normally the HDS and the WPS will short to GND together. However if the card corner is cut, the WPS operation is delayed momentarily because the card edge takes a tiny bit longer to reach the ball. The CRC detects that the WPS was not connected to GND when the HDS connected to GND and thus determines that the card is write protected. In this situation, even though the card passes through the calculator, the write process in inhibited.



Contact operating balls Contact fingers



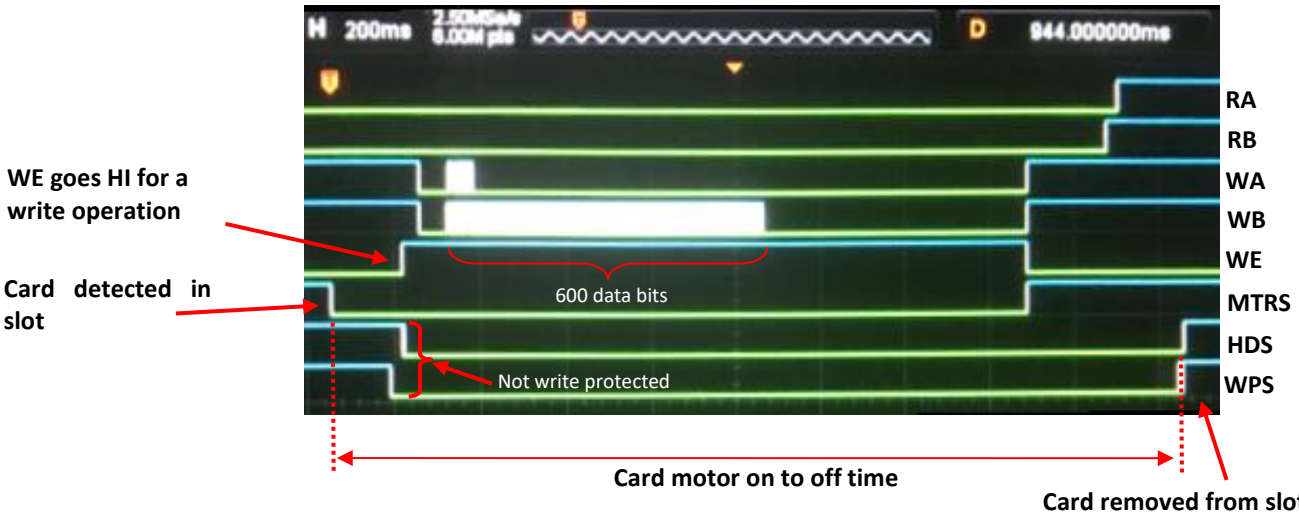
Circuit board mating contacts

Side 1 is write protected

Magnetic card

Read/Write head

The trace below shows a small program being written for a HP-65 to a non-write protected card. The PRGM/RUN switch is in the PRGM position.

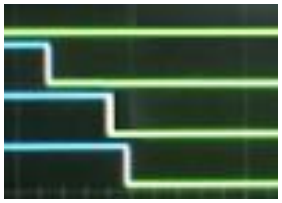


When the card entered the slot, the MTRS switch closed pulling the MTRS line LO. The CRC chip detected the change and turned on the motor to pull the card through the calculator. The WPS switch then closed and pulled the WPS line LO. A short time later the HDS switch closed and pulled the HDS line LO. Because the WPS line was LO when this happened, the CRC chip knows the card is not write protected and it then set the WE line HI. This condition enables the card write circuitry to become active in the sense chip.

As the card passes over the R/W head the CRC send the program data bits to the sense chip on the WA and WB lines. After the 600 data bits are written to the card, the CRC tests for the MTRS to open again. This will happen after the card end passes the nylon ball and releases the MTRS switch allowing the line to return to a HI state. The WE line was released back into a HI state turning off the sense chip write circuits. The WA and WB lines also went HI at this point. Soon after, the card end will pass by the WPS nylon ball and the WPS line will go HI, shortly followed by the HDS line. You will notice that both these lines appeared to go HI simultaneously. This is because the card was quickly removed from the card slot after the motor stopped and the scope could not see the difference in the switches working. If the card is not manually removed from the slot after the motor stops, then the HDS line will stay LO.

If there are any switches that do not operate or they operate outside time limits determined by the CRC circuitry, then the motor will be turned off and an error will be flagged to the ARC chip which will make the LED display flash.

The card motor was on for about 2 seconds. You can see that there is still some area behind the WB data that could be used to store more information. This data could still be written until just before the CRC chip put the WE line back to LO. This would indicate that the card is moving a little bit slow through the reader resulting in data that is bunched up a bit.



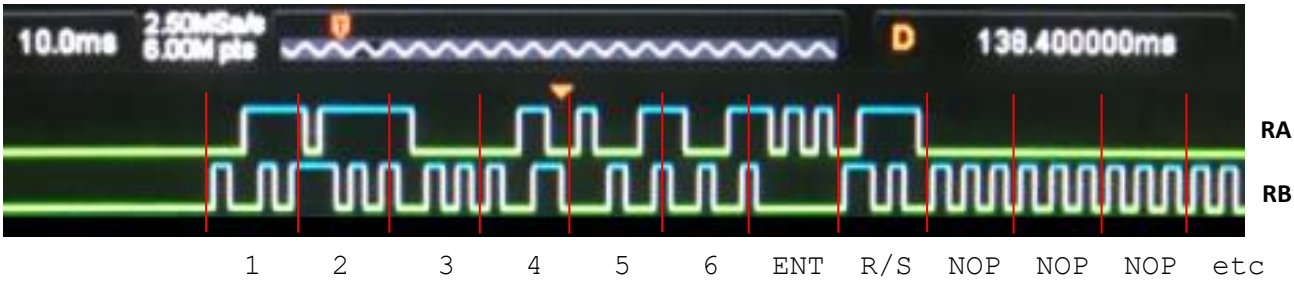
If the card is write protected by cutting the card corner off, then it takes a little bit longer for the WPS to be operated by the nylon ball. This means the HDS will operate first and bring the HDS line LO before the WPS line. The CRC will interpret this condition as a write protected card and will not put the WE line HI. Even if the 600 data bits were transferred to the sense chip, the data will not be written to the card.

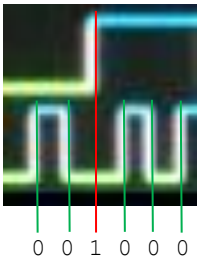
If the PRGM/RUN switch is in the RUN position when the card is placed in the slot, the CRC will initiate a card read process. Regardless of whether the card is write protected or not, the WE line will remain LO disabling the sense chip write circuitry and allow the data to be read from the magnetic strip on the card. The sense chip will amplify and condition the signals from the read/write head and send the data out on the RA and RB lines as shown below. All other switch signals work as in the write process.

This card was recorded with the following small program. Each key press is saved as a 6 bit value.

#	Key	Value	6 Bit Binary
1.	1	4	000100
2.	2	3	000011
3.	3	2	000010
4.	4	20	010100
5.	5	19	010011
6.	6	18	010010
7.	ENTER	62	111110
8.	R/S	34	100010

The data bits that were read from the card are expanded below.





Expanded view of the first 6 bit program code

Each time a data bit is 1, the RA line will change state.
Each time a data bit is 0, the RB line will change state.
This is called a self clocking scheme and helps with data retrieval.

The data is stored Least Significant Bit (LSB) first which is why it looks opposite to the 6 bit binary above. All unused program codes are set to the 6 bit NOP value of – 000000.

The HP-65 relies on timing signals to read and write the 100 program steps (600 data bits) as there is a direct exchange between the card and the program storage memory via the sense chip and controlled by the CRC.

The HP-67 and HP-97 read and write the cards in a different way than the HP-65. These calculators control the card read write process still via the CRC but controlled by the Microcode stored in ROM. This is a much better way of providing card access as it opens the door for more features that the user can utilize such as card merging and storing memory in data cards. This also provides a mechanism for storing information about the calculator settings such as DEG RAD GRD mode which may need to be set to run the program properly. The card information also includes a checksum so that the stored information can be checked after reading and discarded if there is an error.

The storage registers in the calculators are 56 bits wide (14 x 4 bit nibbles). The data on the cards is stored as half registers made up of 28 bit records (7 x 4 bit nibbles) and comprises the following.

1	Status Information
2	Storage Register $2F (47 decimal) upper half
3	Storage Register $2F (46 decimal) lower half
	. . . . .
32	Storage Register $20 (32 decimal) upper half
33	Storage Register $20 (32 decimal) lower half
34	Checksum

Therefore the total amount of data stored on each card is 34 x 28 = 952 bits.

The information above is for a single card program. However, programs may be larger and will not fit on a single card, so the HP Microcode tests the size of the program and if it needs more storage, it will split the program and store the data on two cards. A similar process occurs when data cards are stored. This is the familiar [**Crd**] display you see when storing programs or data like this. It is possible that a program is stored only in the upper half of memory. If this is the case, then two cards will still be used for storage even though the first one will have no program information stored – just R/S key codes.

Card Status Record

This record contains information on how the calculator is configured by the user. See page 273 of the HP-67 user manual. The nibbles are stored in the record as shown below.

N6	N5	N4	N3	N2	N1 N0
1 DATA Side 1	1 1 CARD SET	FLAGS	MODE	DSP	MODE
2 DATA Side 2	0 2 CARD SET	1234	0 DEG	0	0 0 SCI
3 PGM Side 1		3210 (BIT)	1 RAD	-	4 0 ENG
4 PGM Side 2		1=SET, 0=CLR	2 GRD	9	2 2 FIX

Example: 3100222 Program card #1 of 1 card set, No Flags set, DEG Mode, DSP=2, FIX Mode
4081940 Program card #2 of 2 card set, Flag1 set, RAD Mode, DSP9, ENG Mode

The previous program example will fit on a single 67 card.

Step	Key	HP-67 PGM Code
001	1	11
002	2	12
003	3	13
004	4	14
005	5	15
006	6	16
007	ENTER	1B

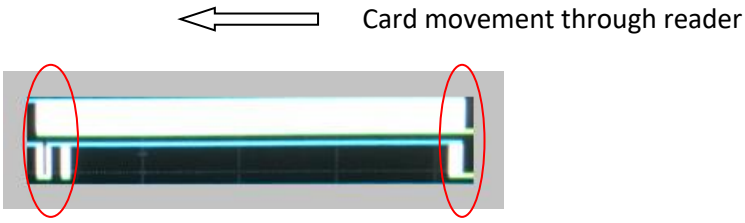
This will be stored in the HP-67 memory register $2F (47 decimal) as:

```
Nibble:    13 12 11 10  9  8  7  6  5  4  3  2  1  0
           1  B  1  6   1  5  1  4  1  3  1  2  1  1
```

If the calculator is set up as in the previous single card Status example, then the data records stored to the card will be:

```
1     STATUS      3100222
2     PGM         1B16151
3     PGM         4131211
       - - -
32    PGM         0000000
33    PGM         0000000
34    CHECKSUM    8D47584
```

The following image shows data on a card that contains the program. The upper trace shows mostly all zeros recorded, which represents the R/S key codes. The lower trace shows some ones recorded on the left and right. The Status and program are on the left of the image and the checksum is on the right.



How the Checksum Is Calculated (Not available for the HP-65)

The checksum for the card data is calculated by adding the sum of all the records 1 through 33.

If you add the records for this card you get

```
Record      Data        Sum
                        0000000
1           3100222     3100222
2           0000000     3100222
3           0000000     3100222
    ----    3100222     records 4 to 31 will all be zeros (R/S keys)
32          1B16151     4C16373
34          4131211     8D47584
```

Note that carries are also added, but not from the last digit. That one is discarded.

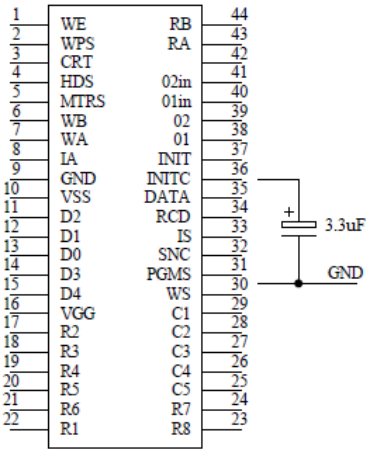
For example:

```
  3100222
+ FEFEFEF
= 2FFF211
```

HP-65 Reset

INIT

VSS



The trace shows how the reset signal (INIT) is held LOW for about 45mS after switch on. This is determined by the 3.3uF capacitor and circuitry internal to the hybrid. It does this so that all the logic circuitry is held in a reset state until the power supply has stabilized after switch on.

This arrangement seems to be typical of the Classics until the HP-67. In this model, the ROMs did not have a reset signal applied to them, only to the CRC chip. In the Spice models, there was no external reset signal at all so the logic IC's probably have internal reset circuits built in.

The card files are text files but have a simple (encoding) method and this was used on some other programs I have done in the past but the method was changed at some point. I added a header to tell the difference.

The first line in the file is "NeWe"

The second line is the character count of the following text, and can be used for initial verify of the contents. The following simple text encryption is each text character XOR'd with 55hex. The following text data is as follows...

Calculator ID:
    67 or 97
Card bitmap file name:
    Must be in same directory as the card file
Card name:
    Limited to ASCII character values 0x20 - 0x7F
    20 characters maximum

Card Data:

HP registers contain 7 bytes or 56 bits.
Card #1 storage is from HP memory $2F -> $20
Card #2 storage is from HP memory $1F -> $10

Each original card has 34 records of data each containing 28 bits of information. This comprises a STATUS record, 32 data records and a Checksum record. This gives a total count of 952 bits per card.

The first of the card file data is 21 0's. (3 records). This is not normally needed, or stored on real cards, but early on I didn't fully understand the way the HP microcode interacts with the CRC. You can ignore this data for reading, but must be inserted for writing if the file format is to stay compatible. For the same reason, the Checksum record is duplicated at the end of the file.

The next 7 nibbles of data are the card STATUS information

For example, if the calculator is set to:

```
FIX 3, RAD Mode, Flag 1 = 1, Single Sided, Program Card
2
2
3
1
8  (4 bits - Flag1 = Set, bit 3 = 1)
1
3
```

The next data is the 112 steps of code.

Consider this data is in register $2F (47dec):

```
DB55DC542D1BFA  (Nibble 13 - Nibble 0)
```

The data for the card file is as follows...

```
5 - nibble 7
C
```

```
D
5
5
B
D - nibble 13
A - nibble 0
F
B
1
D
2
4 - nibble 6
```

Continue same sequence down to RAM address $20 (32dec)

The next data is 7 nibbles of checksum

The checksum is the sum of the STATUS and program nibbles

Example: A 67 program is created as follows:

```
FIX 2, DEG, No Flags, Single Sided, Program Card #1
7 presses of Key 1
```
(Note: The program code for Key [1] is 11)

Register $2F: `11111111111111`

Card data: (Ignoring blank zeros' mentioned above)

```
2220013             STATUS
1111111             Reg $2F
1111111
0000000             memory $2F -> memory $20
0000000
4442235             checksum
```

The checksum is stored in the file as...

```
4
4
4
2
2
3
5
4
4
4
2
2
3
5
```

Notice that the checksum data is duplicated, however as mentioned previously, only one checksum record is required for the actual card.

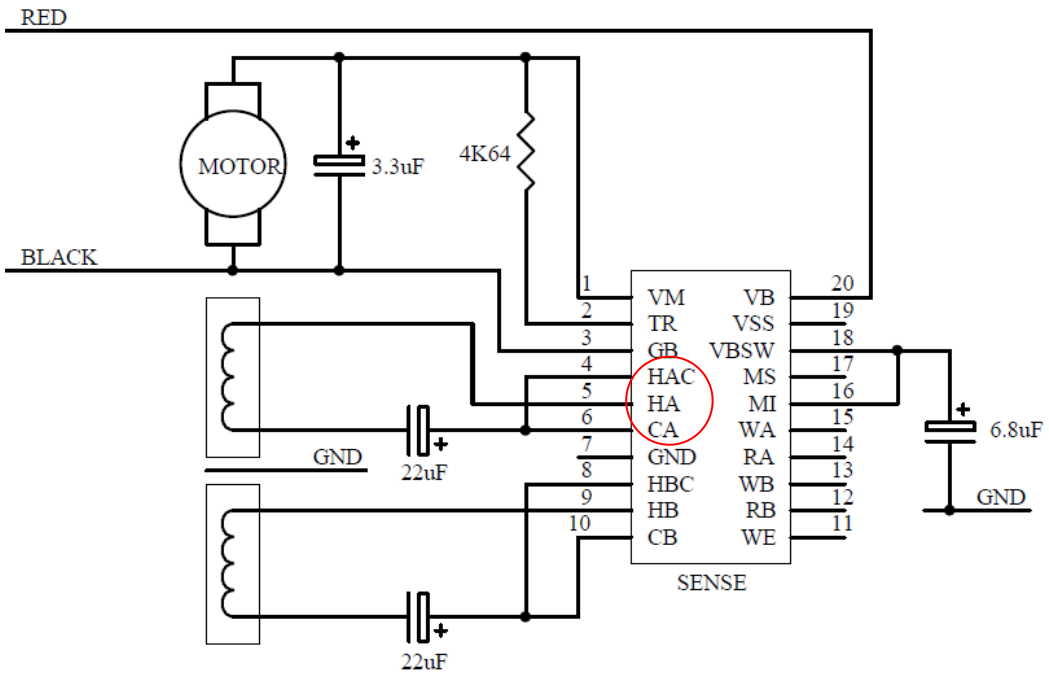
If there is program data for card #2 then that card is stored in the same format but the STATUS bits will be adjusted for Card #2.

Data cards are stored the same except that the memory storage registers are used.

Data Card #1 from RAM memory $0F down to $00
Data Card #2 from RAM memory $3F down to $30

The card reader circuit for the HP-65 and HP-67.

(Note that the 65 and 67 look the same but have different serial numbers. (1826-0158 and 1826-0322)



This trace shows the steady state voltage at the read write head. It shows some power supply hash which is about +/- 200mV.

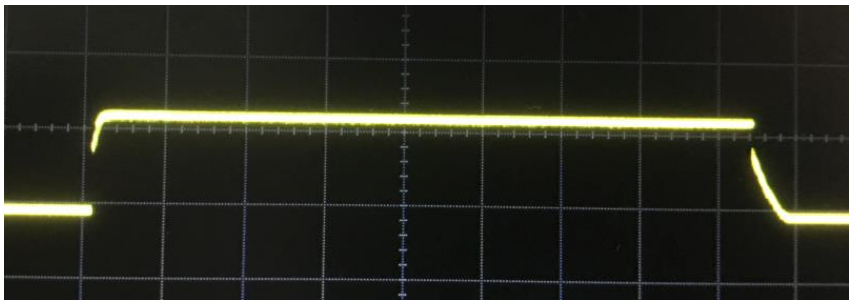
This trace is showing a HP-65 write operation looking at the logic data coming into the WA pin and the AC data being fed into the write head from the sense chip. The vertical scale is 2V/Div. The horizontal scale is 1mS per division showing that the data pulses are about 1.2mS in duration.

This trace is showing a HP-65 read operation looking at the logic data coming out of the RA pin and the AC data coming from the read/write head to the sense chip. The HAC and HA vertical scale is 0.5 V/Div and the RA vertical scale is 2V/Div

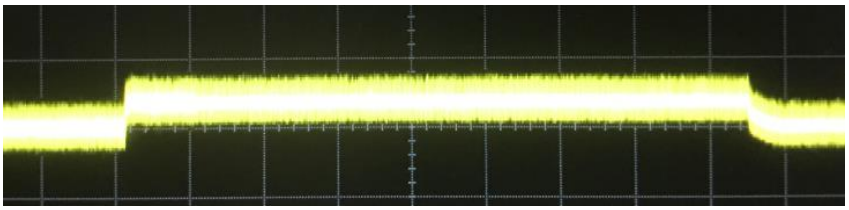
Relative to ground, the negative switching threshold appears to be about 1.3V and the positive switching threshold appears to be about 1.5V. This equates to a 200mV signal.

This trace shows the write enable input (WE) in blue with the CA pin. The initial pulses on the CA pin correspond to the program data pulses being written to the card. Most of the program memory is empty.

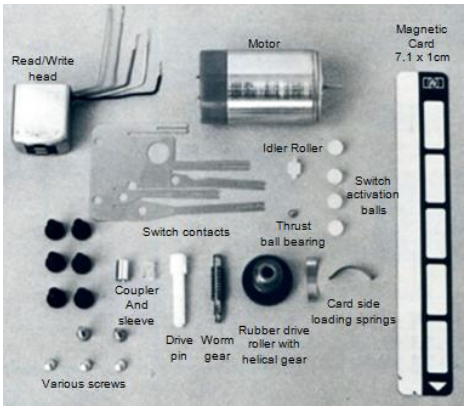
This is a trace of the motor voltage when reading a card is about 2.4V.



This trace is looking at the voltage across the 4K64 trim resistor connected to the sense chip pin 2. It changes from 0V to about 0.5V and shows some power supply hash superimposed on top.



Card Reader Gummy Repair Notes



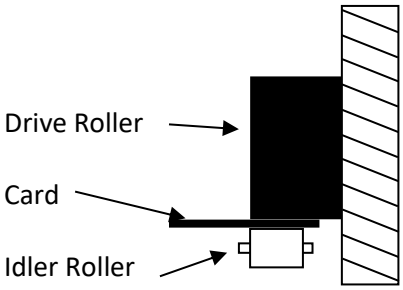
A word of caution when disassembling the card reader. After the motor is removed, do not turn the card reader upside down. The small idler gear is now exposed, and it will easily fall out and probably without your knowledge. Most times this little part is extremely hard to find once this happens.

The material used to grip the cards and enable the drive roller to pull them through the reader does its job quite well. However, over time it deteriorates into a messy blob of goo and the card reader stops working as a result.

There are a few ways to repair this problem but this discussion will be about using easily available o-rings.

There is plenty of information on how to do this type of repair on the web so it is not worth re-inventing the wheel here. The replacement o-rings are 005 types, 7/64” ID and 15/64” OD with 1/8” thickness.

If the drive roller diameter is too wide, as in using an oversized o-ring, then too much pressure will be placed on the card and rollers as it is squeezed between these two parts.



When you press the o-rings onto the drive wheel axle, they can expand outward which increases the diameter to the correct ¼” size.

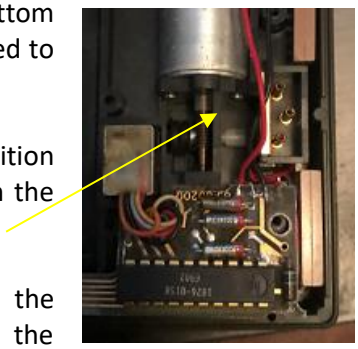
The drive pin that the drive roller spins on is supposed to be eccentric so that there is a small amount of adjustment that can be made to alter the pressure or “grip” on the card as it passes through the reader. After repairing a few, it struck me that I could not figure out how this worked. The drive pin looks like a normal straight pin shaft so turning it would not affect the positioning of the drive roller. Even if it was offset, turning it would result in the shaft end rotating in an eccentric manner, not the shaft itself.

On closer inspection however, one end of the shaft has a smaller diameter which mates with a support hole on the inner surface of the card reader moulding. It is this tiny part of the shaft that is offset slightly and provides the eccentricity and in effect changes the angle of the shaft slightly between the two support holes.

Fig A. Drive Pin looking from the side

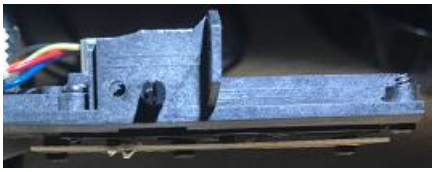
Fig B. Drive Pin looking from the top

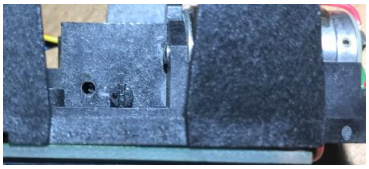
Looking at the position of the offset in *Fig A* you can see that it is towards the bottom of the shaft. When in this position, the drive wheel will have less pressure applied to the card and idler wheel. This is where it should be if the o-rings need trimming.



In *Fig B* you can see that the screw driver adjustment slot is vertical and the position indicator is facing forward. For minimum pressure, this is how it would look in the HP-65 and 67 while looking down at the card reader.

In other words, with the position indicator pointing towards the sense chip, the lightest pressure is applied. If the protrusion is towards the DC motor, then the maximum pressure is applied. There is a procedure for adjusting this pressure in the HP-97 service manual which is freely available on the web.





The HP-41 and HP-97 card readers with the minimum pressure applied

Some early adjusters may not have the small protrusion on the end of the adjuster and it appears that the most or least adjustment is when the slot is in the horizontal position. You could determine which is which and with a permanent marker, mark above the slot for the least position.

Note that when the two o-rings are fitted, the sides of the rings should not extend past the end of the drive wheel shaft. This will cause the drive wheel to be too tight when it is placed in position in the card reader frame. If this happens the sides of the o-rings can be wiped against the flat edge of a file to flatten them a tiny bit. If the o-rings are pressed too hard against the flat edge of the drive wheel, then they can be squashed. If this happens, the outer diameter can increase in size and add to the drive pressure.

Super glue works well to secure the o-rings to the drive wheel. It only needs to be applied to the flat surface of the drive wheel as indicated by the green arrow. Use the glue sparingly. A drop of glue can be placed on a piece of paper and a small pin can be used to scoop some up and apply to the wheel surface. Once the o-ring has been slid into place, a similar amount of glue can be applied to the side of that o-ring and the next one put in place.

With the drive wheel fitted on its axle and without the motor, the drive wheel should spin freely. If not, the o-rings are oversize and are causing drag against the card reader frame. If a card is inserted into the reader, it

should slide in with light friction when travelling through. The pressure should get progressively harder as the adjustment is increased.

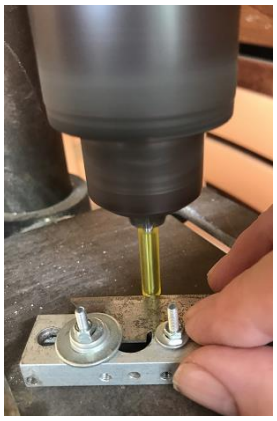
Fuel Tube Drive Wheel Repair

The gummy wheel repair can also be accomplished with some silicon fuel line. I bought some from the following link, but there might be other suppliers that are more suitable.

https://www.ebay.com.au/itm/174332357092

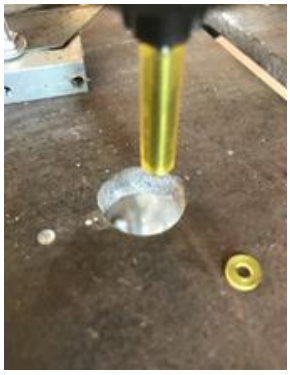
The fuel line can easily be cut to size using general home tools and is easier if you have access to a bench drill. This will help to keep the tube cutting square. I found an old piece of formed aluminium and a few small machine screws and washers to use as a holder for a box cutter blade. This "tool" can be placed on the flat bed of the drill press and easily moved around while remaining at right angles to the drill centre line. If are careful with your fingers, you could omit the tool and just have the box cutter blade lying flat on the drill press table. However, it is a bit more fiddly this way.

I cut off about 30mm of the tubing and slid it onto a 3mm drill from the non-fluted end. It was pushed far enough up the drill shank to leave a small part of the shank protruding from the lower end of the tube. The drill provides support for the tubing.



The drill is then placed into the chuck and secured. Place the box cutter tool on the drill table and move the table up so that the box cutter is a millimetre or so above the end of the silicon tube. Secure the drill table in this position. You could also position the drill table and afterwards lower the drill to the same position and secure the drill bit.

Using care, turn the drill on with a slow to medium speed and slide the box cutter towards the drill and cut through the tubing. You will feel the box cutter blade touch the drill shank when the tubing has been cut through properly. Move the box cutter away from the tubing.



Turn off the drill press and after it stops you can remove the small piece of tubing from the bottom of the drill shank. This ensures that the lower edge or the tube material has a nice clean cut.

Now you need to raise the box cutter blade upwards to the height of the drive wheel shaft which is 0.15 inch. If you cut the tube just a fraction shorter, 0.14 inch (3.5mm), you will guarantee that the tube will not bind in the drive wheel well when it is reassembled. I found a small piece of flat steel with a thickness that is close enough.

The box cutter tool is placed on top of this metal to raise the blade the required distance above the previous cut. Repeat the process to cut the tubing.

The new part can be fitted to the drive wheel with a small amount of super glue dabbed onto the side of the silicon tube where it will butt up against the metal shoulder of the drive wheel.

Again, when reassembled onto the shaft of the reader, the wheel should spin freely. The pressure should be adjusted so that the card enters the reader and has some friction against the wheel. The motor can be reassembled and the reader reassembled into the calculator for final testing. If all is well, it should all work.

The other item that often needs attention after years in service is the coupler that connects the DC motor shaft to the helical gear. This part looks like a small aluminium cylinder between the motor and the helical gear. The inside of the coupler is made out of a material that deteriorates over time and crumbles and in doing so can no longer work in the way intended. With the motor removed from the card reader frame, you should be able to move the helical gear back and forth a small amount. If you gently pull the helical gear away from the motor hopefully it will stay in place. If it slides off the motor shaft easily, then a repair is required. If left in this condition, the motor shaft will spin inside the coupler and the card will travel too slowly through the reader or not at all.

There are a few methods mentioned on the web to fix this issue. The method described here is quite easy and cheap to do. Note before disassembly that the helical gear has some sideways movement.

Note: The card reader has many small parts that can disappear into oblivion if not handled with care. Try to keep your fingers away from the circuit board and its components.

All you need is the following...

- A small length of 75 ohm RG-58 coax
- Stanley (box cutter) blade
- Side cutters
- Pliers
- 5/64 drill bit
- Bench drill
- Small piece of timber about ½ inch thin
- Super glue

This procedure may sound long winded on paper, but in practice is easy to do. Please read through all of the notes before trying it out. It assumes the DC motor has already been removed from the card reader.

Start off by removing the coupler from the helical gear. It might be a bit tight so gentle twisting pressure from some pliers may help. Clean up any remaining coupler material from the helical gear and wipe with Isopropyl

alcohol or other cleaning agent. Keep the aluminium cover as it will be used later on. Put the helical gear and motor screws somewhere safe.

Strip about 3 inches of the outer sheath and underlying wire braid from the RG-58 cable. There will be a translucent inner core remaining with a copper wire in the centre.

Using some side cutters, cut into the translucent material and down to, but not through, the copper wire. The idea is to remove some of the sheath from the copper wire. While turning the wire, repeat this until the inner core can be pulled from the copper wire.

Use the box cutter to trim one end as close as possible at 90˚. It might be worthwhile practising this a bit as it is not that easy to do. This will be the end of the sheath that you need to drill into.

The sheath needs to be centre drilled using the 9/64 drill bit and will ensure that it fits snugly over the end of the helical gear shaft. The hole needs to be drilled as close as possible to the centre of the sheath and ideally this job would be performed using a lathe. Unfortunately not many people have access to one of these so it can be done using a bench drill. If this is not available either, do not attempt to do the drilling by hand. If the hole is off centre, even by a tiny bit, the shaft will vibrate in use and may stop the card reader from working properly and could also damage the motor bearings.

Put the drill in the drill press chuck leaving about 1 ½ inches exposed. Move the drill press table so that it is about 2 inches below the drill tip. The table usually has a hole in the middle so make sure the drill can pass through this. Tighten the table securely so it won't move.

Secure the piece of timber to the drill table. Using one or two bolts through it and the table is preferable to using clamps.

Drill a 9/64 hole straight through the timber. While the drill is spinning, you should move the chuck up and down a few times to make sure the hole is nice and clean. Turn the drill off.

From this point on, the table and timber cannot move is relation to the drill centre line. If it does, even a tiny bit, you will have to start the procedure over from scratch.

Undo the chuck and open it wide enough so the drill can be removed without upsetting anything.

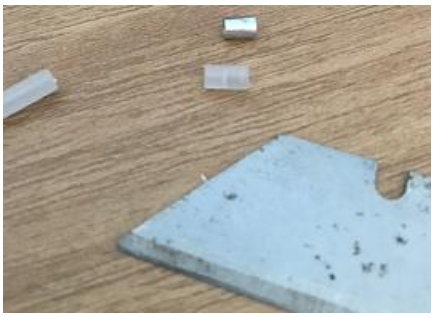
Turn the drill tip side upwards and push it up through the hole from underneath the timber. It will be a tight fit (which is good) so don't be tempted to use a hammer or similar to tap it in as this could move the table or timber. You can push the drill upwards using the side face of some pliers or grip the drill shaft with the pliers being careful not to bend or break it.

Insert the translucent sheath into the chuck with the trimmed edge facing towards the drill. Tighten the chuck but only finger tight. It should be tight enough so the sheath won't slip but won't be crushed also.

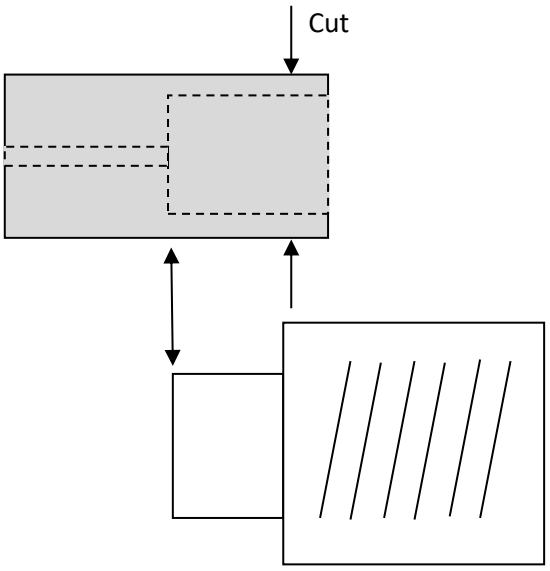
Start the drill and move the chuck down so that the drill bit penetrates about ¼ inch into the sheath and bring the drill back out. If all is well you should have a nicely centred hole in the sheath.

Remove the sheath from the chuck and using the box cutter blade, cut of the end that was drilled to a length of about ½ inch.

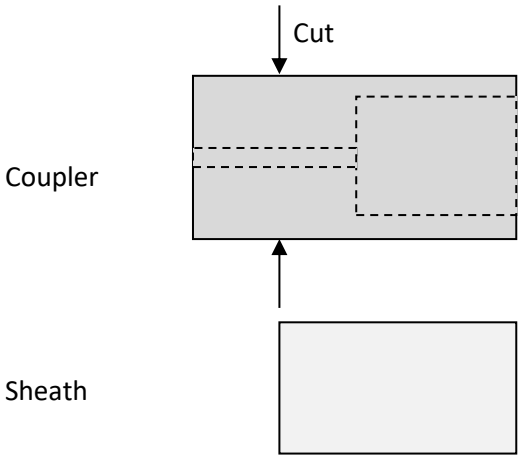
If you want to, you can now repeat the process and make a few more coupler parts for spares.



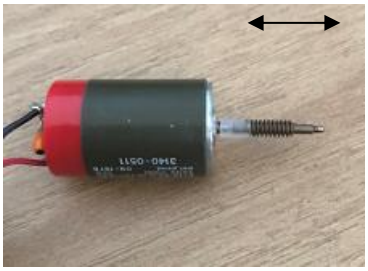
You should be able to see how far the drill bit went into the coupler. Place the coupler beside the helical gear and line it up so the bottom of the drill hole lines up with the end of the helical gear. Place the box cutter blade at 90˚ across the coupler at the point where it lines up with the shoulder. This is where the earlier practice will come in handy.

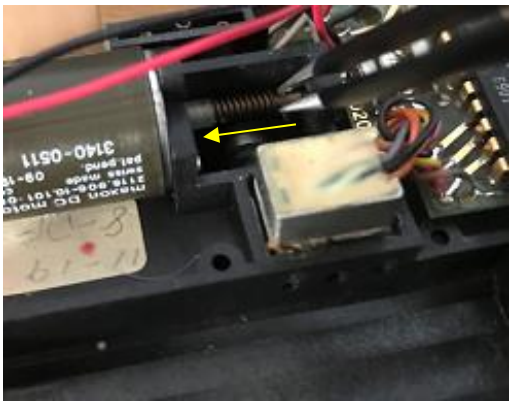


Once this has been cut to match, line up the coupler with the original aluminium retainer sheath and trim the end of the coupler to match its length.



If you make a mistake cutting the new coupler to size, use another of the spares and start again. If the drill is still set up it will be easy to make some more couplers. Without glue, put the coupler onto the helical gear end and push the coupler all the way onto the motor shaft. You should notice that the motor shaft has some back and forth movement. This is ok.





Insert the motor back into the card reader frame and make sure the tip on the right side of the helical gear is aligned to the bearing end in the card reader frame. Insert and tighten the two motor screws just enough to hold the motor in place. You will probably find that the helical gear will not move back and forth now. You can use some needle nose pliers or a flat bladed screw driver to gently push the helical gear and coupler closer to the motor.

Keep tightening the motor screws and moving the coupler making sure the helical gear always has some back and forth movement. If not by the time the motor is fully tightened, you will have to remove the motor, trim the motor side of the couple a bit more and then reassemble. If this shaft has no side play because the coupler is too long then the motor shaft will be pushed too hard into the motor casing and could damage the bearings and other motor internals.

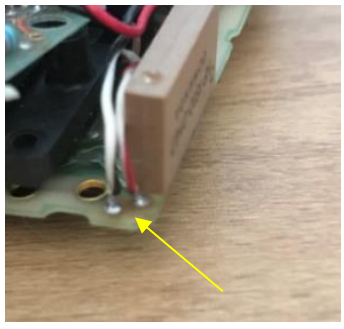
When satisfied that the coupler is ok, disassemble it and place a small amount of super glue around the helical gear drive end where the coupler fits, being careful not to put too much on. Overdoing the glue may cause the glue to dribble into the centre hole or out over the gear. Insert the coupler onto the helical gear.

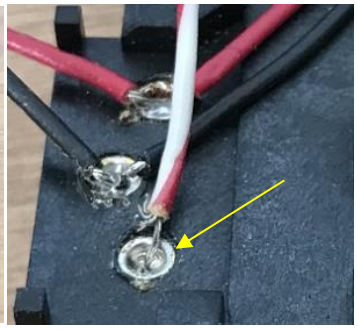
Place the gear and coupler onto a flat surface and roll it with light pressure along the length of your finger. This will help centre the coupler onto the shaft. Leave it alone now while the glue sets. Note: There is no need to glue the coupler to the motor shaft.

You need to clean the helical shaft thrust bearing which is a tiny ball bearing pressed into the card reader assembly. After cleaning, a tiny dab of thrust bearing grease should be applied for lubrication and helps with quiet operation. This grease can be found in small tubes at hobby shops.

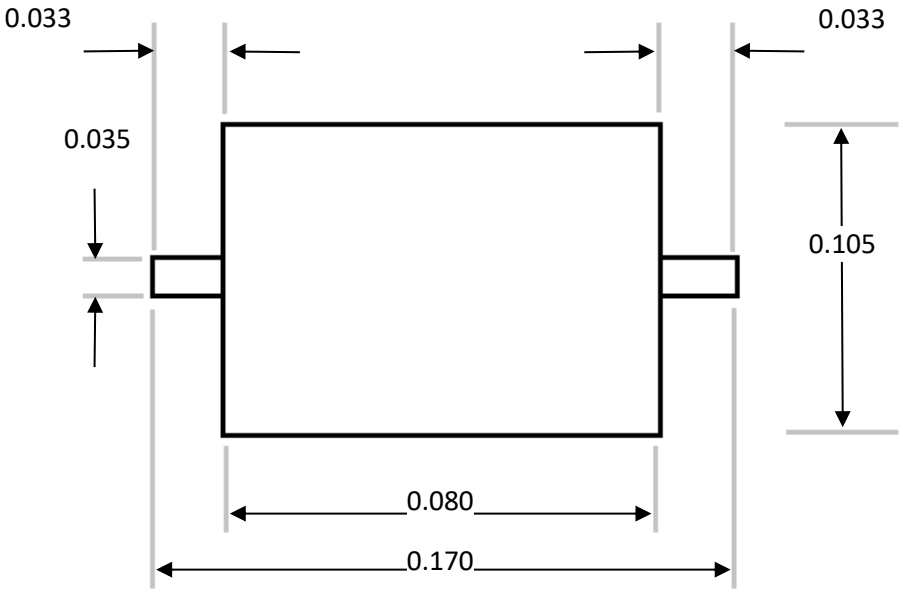
When the glue has cured properly, slide the coupler onto the motor shaft but this time **stop about 3mm before the coupler touches the motor end**. Reassemble the motor and while tightening the screws, the end of the shaft will press on the thrust bearing located in the card reader housing which will push the coupler further onto the motor shaft. After assembly you can carefully move the shaft a **tiny** bit towards the motor using a flat bladed screwdriver in the helical gear end. This will leave a miniscule amount of side play but too much side play will result in noisy operation with the shaft vibrating back and forth. See here for the HP-97 service manual. There is some card reader info near the end of the document.

When fiddling with a disassembled card reader it is possible that some of the wiring gets stressed from all the movement. This is especially so where wires are soldered to the circuit boards. These cables are made up of many tiny wires which can individually break. This may limit the amount of current that can pass through them and could upset the calculator operation. Check the wiring at these points to make sure some or all of the wires have not broken. If any have you will need to re-solder the connection.



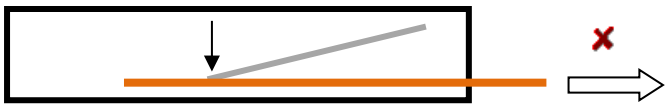


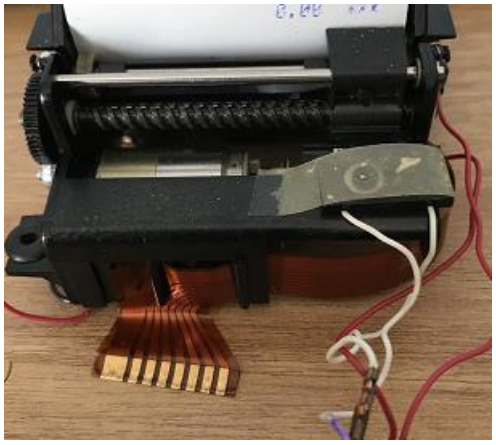
Idler Roller Dimensions



Printer Ribbon Removal Tool

Sometimes it is necessary to remove the printer from the HP-97 calculator. The socket where the ribbon cable connects to is a captive socket. This means it will slip in relatively easily but it won't come back out. This is because the socket contacts are angled in such a way as to bite down on the ribbon cable if it is pulled. To remove the ribbon without damaging it a special tool is required. This tool is not available to buy but making one is easy.

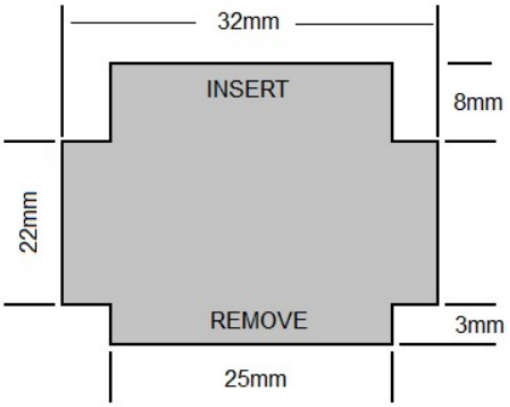




I used some plastic cut to size from one of those cheap disposable food containers. It is rigid enough for the job and just thick enough. Metal could also be used if it was the same thickness and all rough edges removed.

The dimensions of the tool are shown in the drawing

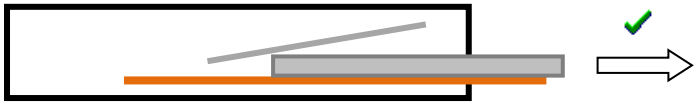
.



If you look at the printer circuit board from the component side, the contacts which grip the cable are located on the socket top side furthest from the circuit board. It is the ends of these contacts that bite down and grip the ribbon cable and won't let it come out.

To remove the ribbon cable, insert the tool end marked "remove" into the ribbon cable socket between these contacts and the ribbon cable. The shoulders on the tool only allow it to be inserted about 3mm into the socket. This has the effect of pushing the contacts away from the ribbon cable but it is not long enough to extend beyond the end of the contacts.

The ribbon cable can now be removed.



When it comes time to reinsert the cable, push the tool end marked "insert" into the fold in the cable and push the cable back into the socket.

This tool can also be used for other printer models like the HP-19C.

Cleaning Keyboard Contacts

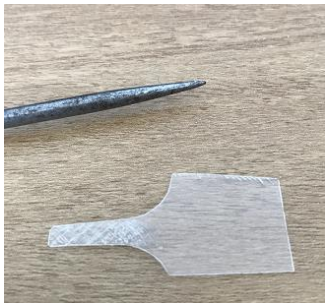
The HP Classic calculators are getting on for 50 years old now and as such, some of the parts that get used a lot are breaking down. The keyboard contacts are one of these items. The contacts are made out of tinned copper alloy and work on the “cricket” toy principle where applying increasing force to the raised dome will eventually cause it to give way and make contact with the circuit board beneath. One released, the contact will spring back to its original rest position. That is of course unless it has failed in some way. Two ways it can fail are through cracking of the material or excessive use has reduced the tensile strength of the material and it won’t spring back properly. In either case the keys begin to feel weak spongy in use.

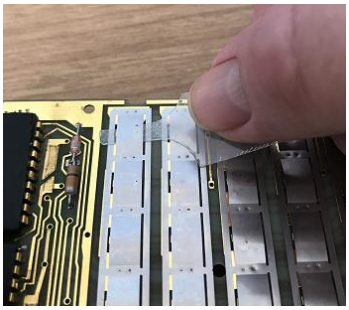
One other problem if that the contact between the circuit board and the dome becomes contaminated with corrosion or from rubbish that has accumulated there over the years – dust etc. This is usually detected by the user when the keys fail to respond properly when pressed. Sometimes they work and other times, multiple key presses are required which can result in the calculator thinking the key was pressed many times instead of just once. Luckily in most cases, the circuit board contacts were gold plated and this is resistant to corrosion. The key domes are not and can corrode under the right conditions.

Cleaning the key contact is straight forward once the calculator has been stripped down. The items required for this are isopropyl alcohol and a tool for cleaning. You can also use commercial contact cleaners but these can be expensive. Some advocate spraying contact cleaner directly into an unassembled calculator in the hope it will clean everything. This is unlikely to get into the contacts where it is needed as there is a plastic membrane between the plastic keys and the contacts, so a strip down is the best method albeit a bit more time consuming.

The distance between the key domes and the circuit board is only a few millimetres so a cleaning tool needs to be thin, narrow and flexible. It is best if the cleaning surface is rough instead of smooth but not harsh enough to damage the keyboard parts. I use a piece cut from a plastic disposable food container for the tool. These are cheap, thin, flexible and relatively soft.

The tongue of the tool measures 4mm x 15mm and the finger area is about 25mm square. After cutting out a suitable piece for the tool from the food container and before cutting the tool to size, use a nail or a scriber to cut a cross hatch pattern on either side of the plastic where the tough area will be. This is the rough surface that will help remove any grime under the contact.





The tool can be slid in between the contact dome and circuit board which has a drop or two of isopropyl alcohol applied and then scrubbed with a back and forth motion with light finger pressure pushing down on the dome. Care is required not to lift the tool upwards or damage to the dome might result.

When finished, give the top of the keyboard another wipe with a clean cloth and isopropyl alcohol. Always wipe along the length of a key dome strip, never across it.

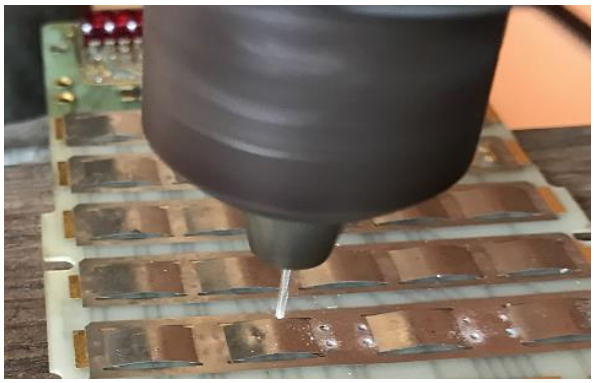
Repairing a Damaged Key Strip

If you have a non-working calculator that you can use for spare parts, then it is possible to transplant a good key strip to repair a working calculator with the faulty strip.

Classic key dome strips come in two varieties. One with 5 domes (the top 3 strips) and one with 4 domes (the bottom 5 strips) and they are not interchangeable. The key strips are spot welded onto the circuit board and because of this the strips cannot be removed without these spot welds being separated.

If you try to break the spot welds by levering the strips away from the circuit board with a screw driver, then in most cases the strips will either deform or tear. If this happens you will most likely never get a good quality key "feel" after transplanting it. Getting a good "feel" requires that the strip be in good condition and that it lies flat on the surface of the circuit board help in position at the same mounting points as original.

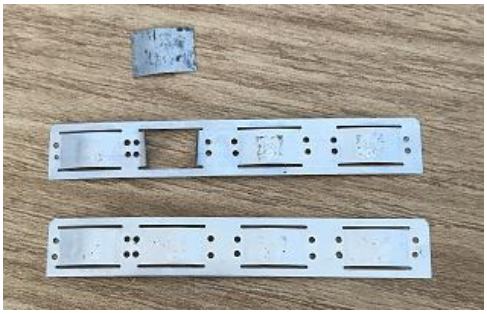
The method described here to remove a strip requires a drill press and a 1mm drill bit and a soldering iron. To make sure the connecting pins at the end of the circuit board do not get damaged, it is wise to support the board with something like a piece of timber that is thicker than the pin height.

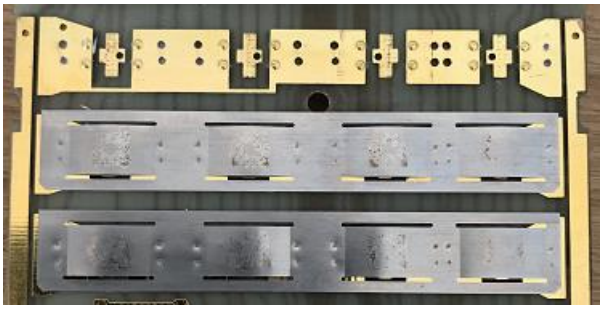


The drill is used to drill through the spot weld but not through the circuit board. You need to drill the spot welds from the donor board and the ones in the faulty strip on the recipient board. Generally once the drilling has commenced, as soon as the drill goes through the strip metal, the strip will lift up a bit and you know it has gone through and you can stop drilling further down into the board. Be careful with the last hole as the drill bit may catch the strip and it will start spinning damage it. You may notice spot welds that have already broken apart, drill these anyway on the donor strip as you will still need to use the holes to connect the strip to the recipient board.

Once the strip has been removed, you might see small burs on the drilled holes. These need to be removed or the strip will not sit flat on the circuit board. You can use a slightly larger drill bit placed onto the hole and rotate it with your fingers to deburr it.

Once the strips have been removed, clean the underside and also the gold plating on the board with isopropyl alcohol. The small crosses in between the larger gold pads are the actual key contact surfaces that mate with the domes.





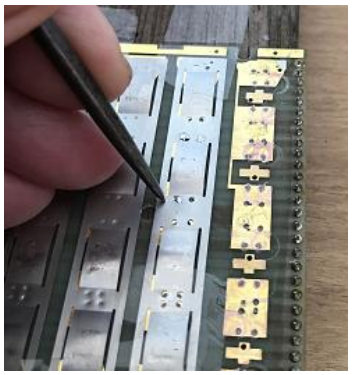
Before attaching the donor strip, it is best to hold the circuit board steady. I used some sticky tape to adhere it to the backing board.

The strip can be manoeuvred into place and you should see some gold under the strip through the holes. If not, you may have to re-drill some of them or else there is nowhere to solder to. “Some” gold through the hole is ok.

Each strip end can also be held in place with tape. Make sure the strip is parallel to the other strips and has the same spacing between.

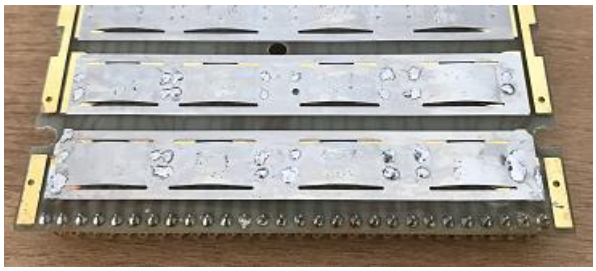
Now comes time for soldering. Make sure the iron is hot and has a medium point tip. The tip cannot be too large or flat or you will inadvertently solder the dome area which will damage the key “feel” later.

Start in the middle of the strip and work out towards the ends. This will help keep the strip flat. The strip will get hot near the iron so soldering away from the tape will stop melting it.



Place the end of the solder over the small hole in the strip and then use the tip to melt the solder and flux directly to the strip hole. Hold the iron in place until you are certain it has soldered. The solder should make a tiny pool once is has soldered properly. If the tip is too small or the iron is not good enough the heat will transfer to the board underneath and the solder may not melt properly.

Once soldered, it is possible that the strip will lift as you remove the iron. I used the pointy end of a scriber to hold the board down while the solder cooled.



Once the strip has a few holes soldered, the tape at the ends can be removed. Continue soldering outwards towards the strip ends trying to make sure the strip stays flat on the board. Try not to flex the board while soldering the strip or it may stay warped after you finish soldering. The may cause faults when the calculator is used, especially for HP-65 and HP67 boards that rely on specific distances between the board and the card sense contacts.

Give the board a good wipe with isopropyl alcohol and the board can be reassembled.

The repair is not guaranteed to feel like a new calculator. That depends on the repair quality and the state of the new key strip which may be worn to some degree. Even so, it should be better than using a damaged or broken strip.

Replacing a Card Reader Side Load Spring – HP-65 HP-67

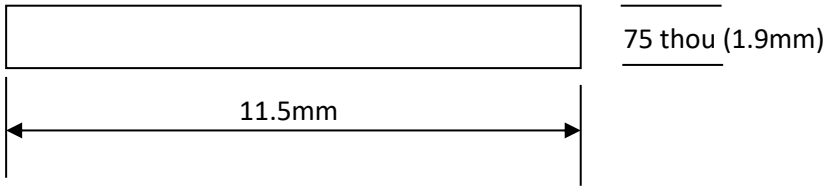
The card side load spring is a part of the card reader mechanics and its job is to allow the card to move through the reader parallel to its motion by keeping sliding up against some parallel walls inside the reader.

There are two of these inside the reader and if one or both break or are missing the cards may wobble back and forth as they move through resulting in data corruption.

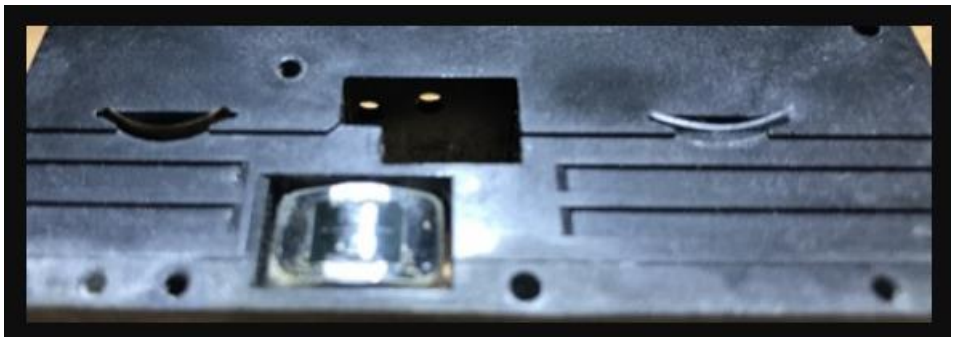
Like the small switch activation balls, the nylon roller and the springs are small and difficult to spot by themselves. They also tend to fly away if they get a chance and become very difficult to find. The nylon balls are 1/8th inch diameter and should be able to be purchased in hardware or bearing outlets. The nylon roller and springs are specialty items and can only be replaced from a donor calculator or by manufacturing them. The roller will need a lathe to accomplish this but the springs can be made out of some materials lying around the house.

For the springs, if spares are not available they can be made with some plastic from packaging for a tooth brush for example. Most items these days are packed on a cardboard backing with formed clear plastic on top, so I guess just about anything like this will do. The only thing is that it can't be too thick or it won't be flexible enough. It can’t be too thin either or it will be too flexible. Aluminium from a soft drink can might be able to be used but it doesn’t have much tensile strength. This means if it bends a bit, it won’t spring back to the original shape very well. The thin plastic is pliable and does return to its basic shape if deformed, especially in this application.

The plastic I used was about 0.006" thick and as mentioned is probably standard on a lot of packaging. The dimensions for the spring material are shown in the drawing.



It is more likely that a piece of this size will have to be cut and trimmed by trial and error. The main thing to note is that it should be cut as parallel as possible so its sits nicely in the areas provided inside the card reader. You should find that you can slide one end in place while holding with tweezers and then bend and feed the other end in place. From there it should sit in place. Because the plastic is springy, it will be really easy for it to fly off somewhere so care will be required while handling. One last thing to note is not to make it so long and tight in the mounts that it won’t flex when the card rubs up against it. It is easy to slide a card in the trough area and test the new spring.

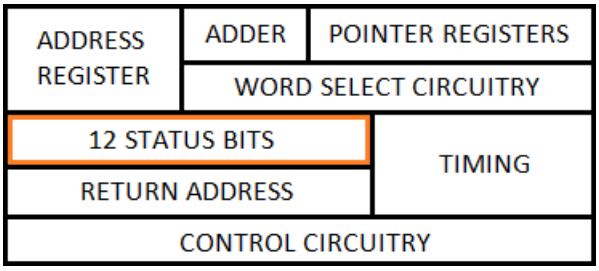


Classic Status Register Hardware Flags

HP-35, HP-45, HP-55, HP-70, HP-80 and HP-65

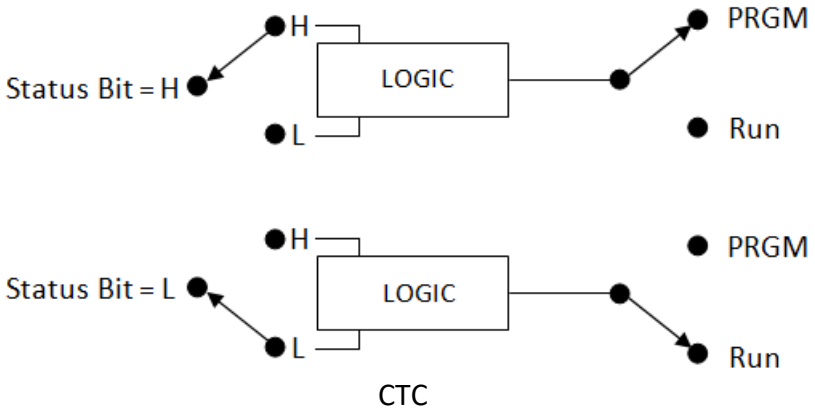
The Status Register is made up of 12 bits which represent True/False flags for the software to use. These can help the software keep track of what is happening during the code execution.

The Status Register is located inside the Control & Timing Circuit. (CTC)



HP-45

Some of these bits reflect the state of the externally connected keyboard or the Run/PRGM/Timer switch because these switches are hard wired to certain bits in the Status Register. The corresponding bit is normally cleared before a test takes place, then the bit can stay True depending on the switch position. The software running inside the calculator uses the state of these bits to determine if a key is being pressed, or for example, if the slide switch is in the Run, or PRGM position after the Status bit is cleared.

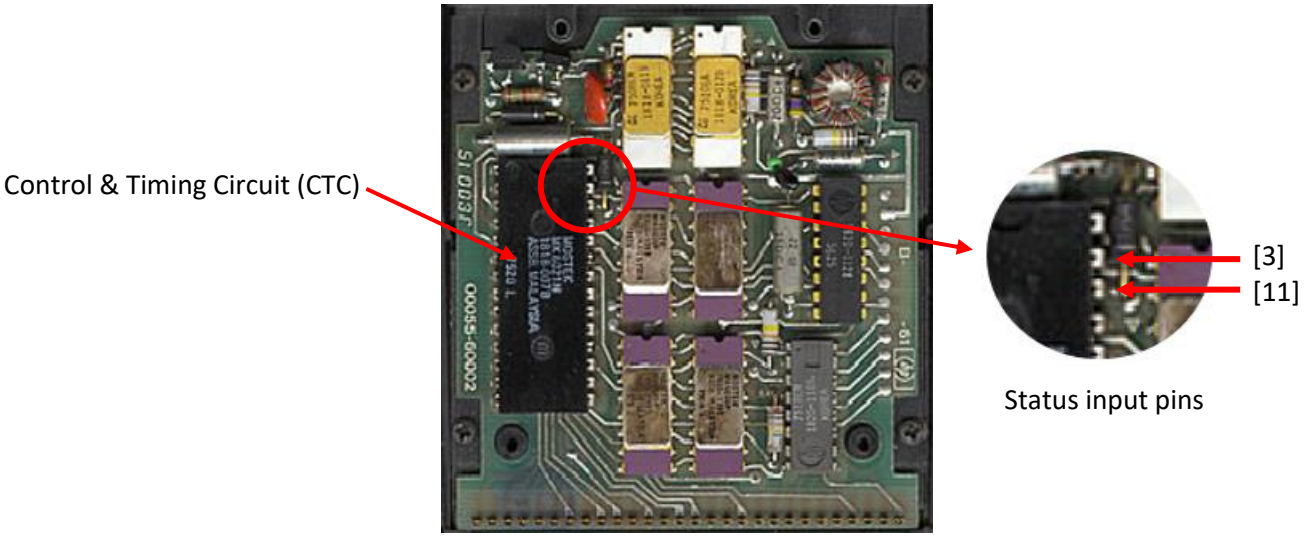


CTC

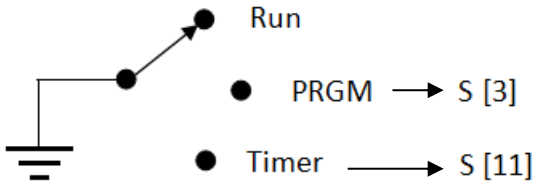
For all Classics, Status bit [0] is connected to the keyboard scanning circuitry inside the CTC chip. This bit is set to 1 (True) if a key is pressed on the keyboard, or if no keys are pressed, then Status bit [0] is set to 0 (False). This is part of the microcode for the HP-45 that detects a key press.

```
dsp3:     1 -> s8                  Key Loop Start
          if s5 # 1                Error Display ?
               then go to dsp5     No
          c + 1 -> c[x]            Yes, flash display on and off
          if no carry go to dsp2
dsp4:     display toggle
dsp5:     if s0 # 1                test Status[0] = True, keypress?
               then go to dsp3     no
dsp8:     0 -> s0                  yes, clear status[0] (stays True if keys are still pressed)
dsp6:     p - 1 -> p               debounce keypress
          if p # 12
               then go to dsp6
          display off              turn off display
          . . . . .                process keypress
```

For the HP-55, there are two extra Status bits that are used for detecting the state of the Run/PRGM/Timer switch position. These are bits [3] and [11] and are connected to the outside world via pins 26 and 27 of the CTC.



HP-55



Status Register	**[11]**	**[3]**
Run	False	False
PRGM	False	True
Timer	True	False

The HP-65 only uses Status bit [3] for detecting the position of the Run/PRGM switch.

Status Register	**[3]**
Run	False
PRGM	True

Status bit [11] in the HP-65 is hardwired to a special Flags register inside the CTC. This register is used for various purposes and if any of the 8 bits of this Flags register change from a True to a False state, then Status[11] is set to True. This is part of the HP-65 code that is testing if Flags[0] was set to True.

```
0 -> s11              reset Status [11]
0 -> f1               set Flags [0] = False
if s11 # 1            was it True previous
 then go to rcl22     no
c -> stack            yes
. . . .
```

Woodstock

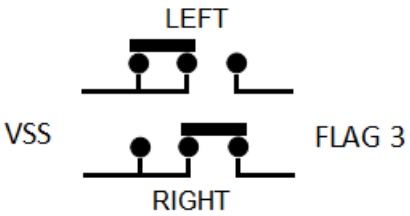
The Woodstock models, including HP-67 have 16 bits in the Status Register which is located in the Arithmetic Timing and Control (ATC) chip.

Status Register	**[15]**
All Keys Up	False
Key Down	True

This is the Flag #1 input to the ACT chip (Pin 3). It is connected to the Cathode Driver (Pin 7) which changes the state of this flag depending on whether a key is pressed or not.

Status Register	**[3]**				
HP-21	HP-22	HP-25(C)	HP27	HP-29C	
Deg	End	PRGM	N/C	PRGM	False
Rad	Begin	Run	N/C	Run	True

This is the Flag #2 input to the ACT chip (Pin 4). The HP-27 does not have a function switch, so this pin is not used. For all the others, the pin is connected to the switch.



In the HP-67 and 97, Status [15] detects a keypress, and the Flag #2 input, which is Pin 4 of the ACT chip, is connected to Status [3]. This flag has many functions and is controlled by the Card Reader Controller (CRC) flag output, pin 27. This output can only be set = True by the CRC after it receives certain instructions. The microcode resets this flag to False for testing purposes, and then it issues an instruction to the CRC. If the result of the instruction is True, the CRC will respond by setting the Flag output to True.

To access the CRC Status [3] flag, the microcode usually executes similar code to this…

```
0 -> S3              clear the Status [3] flag
CRC 100              ask the CRC if it is in a ready state
if S3 = 1            test the resulting S[3] flag state
  then goto xxx      1 = ready
. . . .              0 = not ready
```

There are many microcode instructions sent to the CRC chip to control the card reader. The CRC chip also has its own flags that can be set or cleared. These can then be tested by the microcode for various purposes.

		False	True	Notes
CRC 0060	Set Flag [display digits] = True			No Flag test
CRC 0100	Tests is the CRC is ready	Not Ready	Ready	
CRC 0160	Test Flag [display digits]	Not set	Set	Flag is reset = False
CRC 0260	Card reader motor = ON			No Flag test
CRC 0300	Test if switch = PRGM	Run	PRGM	
CRC 0360	Card reader motor = OFF			No Flag test
CRC 0400	Set Flag [Key pressed] = True			No Flag test
CRC 0500	Test Flag [Key pressed]	No key	Key	Flag is reset = False
CRC 0560	Is a card inserted?	No	Yes	

CRC 0660	Set CRC = Write Mode			
CRC 0760	Set CRC = Read Mode			
CRC 1000	Set Flag [default keys] = True			Keys A – E execute default pgms
CRC 1100	Test Flag [default keys]	No	Yes	Flag is reset = False
CRC 1200	Set Flag [merge]			
CRC 1300	Test Flag [merge]	No	Yes	Flag is reset = False
CRC 1400	Set Flag [pause]			
CRC 1500	Test Flag [merge]	No	Yes	Flag is reset = False
CRC 1700	Is read/write data valid	No	Yes	

The CRC chip also has another four internal flags connected to switches and are used with the HP-97.

CRC Test F1	Key Pressed	No	Yes	
CRC Test F2	Printer Switch = Manual	No	Yes	
CRC Test F3	Printer Switch = Norm	No	Yes	
CRC Test F4	Is Keyboard E line active	No	Yes	Used for detecting SST

The Status[3] bit is also connected to the Printer Interface Keyboard (PIK) chip and operates with the instructions for this chip. HP-91, HP-97, HP-19C.

PIK1120	Printer HOME	S3 = 1 = Yes	
PIK1220	Out Of Paper	S3 = 1 = Yes	
PIK1320	Keypress in Buffer	S3 = 1 = Yes	Buffer holds up to 7 keys
PIK1660	PIK Alpha 6 bit data	S3 Not used	
PIK1720	PIK Numeric 4 bit data	S3 Not used	

The Spice series has Status [15] for detecting a keypress.

Status Register	**[15]**
All Keys Up	False
Key Down	True

These calculators also have a switch and uses Flag [3] for testing.

Status Register		**[3]**		
HP-33E/C	HP-34C	HP-37	HP-38E/C	
PRGM	PRGM	End	End	False
Run	Run	Begin	Begin	True

HP-19C

Status bit [0] is actually an output from pin 22 on the HP-19C ARC chip. It internally controls the Status [15] bit and it also controls some external transistor logic that is connected to the Status [3] bit. The reason for this setup, is that the ARC has a limited amount of Flag inputs from which to test various hardware states and the HP-19C needs more.

Status [0] = False	**[15]**	**Status [0] = True**	**[15]**
All Keys Up	False		True
Key Down	True		False

I'm not 100% sure, but I suspect this arrangement helps to de-bounce the SST key when it is pressed and released.

When Status [0] is set to True by the microcode, the Run/PRGM switch position can be tested.

Status Register	**[3]**
PRGM	False
Run	True

The Printer Mode switch is detected through the same mechanism as a keypress detect because the switches in the HP-19C are connected to the ACT inputs that normally connect to the keyboard. These inputs can be transferred to the A register nibble [1] with the `Keys -> A` instruction. The microcode will then test A[1] for the returned value to determine what mode the printer will work in.

Printer Mode	**A[1]**
Trace	1
Normal	2
Manual	4

Status [3] is also used as mentioned above when testing the CRC and PIK chip responses.

IF … GOTO

Normally, the GOTO instruction can only do an unconditional code jump within the current 256 word ROM Page. This is because the 10 bit instruction can only use the lower 8 bits for the new PC address. The upper 2 bits are for the instruction encoding. To make coding more flexible and allow code jumps for up to 4 ROM pages, new logic was incorporated into the ARC chips for the HP-67, HP-97, Woodstock and Spice calculators, and also my version of the HP-10 microcode.

This new arrangement allows the full 10 bits of the GOTO instruction to be used as the new PC address. In this case, there is no GOTO instruction encoding, it is implied.

How this works is quite simple. If any of the instructions start with “IF” then the SYNC pulse is not generated when the next data is being fetched from ROM. This instruction it is then treated as a GOTO with the 10 bit ROM code used as the PC address.

Examples from HP-19C microcode:

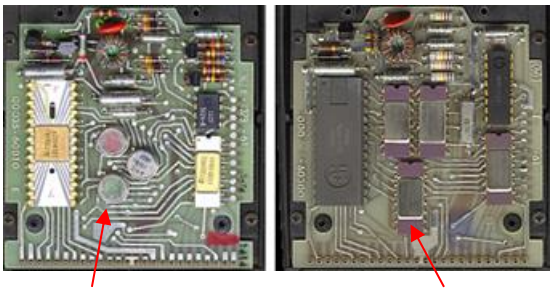
```
$01F7  if s4 = 0                      << if S[4] does = 0
$01F8         then go to $1FD         << the ROM code $1FD becomes the 10 bit
$01F9  if s6 = 1                         GOTO address
$01FA         then go to $1FC
$01FB  c + 1 -> c[p]
$01FC  c + 1 -> c[p]
$01FD  jsb $176                       << this instruction will execute next


$04B4  b exchange c[w]
$04B5  if b[m] = 0
$04B6         then go to $61C         << from ROM Page 4 GOTO ROM Page 6
. . . .                                  address $61C

$061C  delayed select rom 2
$061D  go to $2FC
```

ROM Addressing

Each ROM in the Classics is made up of 256 10 bit words. The ROMs are turned OFF unless they are specifically turned ON during microcode execution. The Program Counter (PC) is reset to zero and ROM 0 is automatically turned ON when the calculator is turned on so that it executes code starting at ROM 0 address $00.

The PC is only 8 bits wide in the Classics and that means it can only be in the range $00 to $FF. If the PC is $FF when it increments, than it will wrap around to $00.

To move the code Program Counter between ROM pages, there are instructions like Select ROM N, or Delayed Select ROM N, where N is the ROM Page that the code needs to jump to. For the HP-35, there are only 3 ROMs available so N can only be 0, 1 or 2. There are 8 ROMs available in the HP-45, however, unlike the HP-35 which had 1 ROM per integrated circuit chip, the HP-45 had 4 ROMs per chip. These chips were referred to a Quad ROM. See: HP-35 ROM.



HP-35 ROM HP-45 Quad ROM

If the Program Count (PC) is at ROM address $02 (hex) when `Select ROM 6` executes, then the PC is incremented by one, and ROM 6 becomes active.
If you look at the initial code for the HP-45, at ROM address $001, the select ROM 6 instruction causes ROM 0 to switch OFF and ROM 6 to switch ON. The PC will now increment to $03 and now all instructions are fetched from ROM 6..

```
$000   pwo1:     jsb pwo2
$001   tms5:     select rom 6
$002   tms2:     jsb frac
```

This is the actual HP-45 code starting in ROM 6.

```
$000   factz4:   go to fact
$001   err2z1:   go to errr
$002   tdmsz0:   clear status     < This is the next instruction to execute
$003             go to tdmszj
$004   oflw:     c + 1 -> c[xs]
$005             jsb fst1
```

The Select ROM mechanism would have created some difficulties for the original programmers. Imagine when the HP-45 was about to be introduced and an engineer found an error in the code at ROM 6 address $00. To fix the error, an extra instruction had to be inserted at address $01. That seems easy to do, just put it in the code and reprogram the ROM chip. Unfortunately, it is not that simple. After inserting the new instruction at address $01, the code would look like this. Unfortunately, when the `select rom 6` instruction executes from ROM 0 address $01, address $002 in ROM 6 does not have the same instruction it was supposed to, it is now at address $03, and the code will not function properly.

```
$000   factz4:   go to fact
$001   nerr2:    go to err5       < inserted code
$002   err2z1:   go to errr       < This is now the next instruction to execute
$003   tdmsz0:   clear status
$004             go to tdmszj
$005   oflw:     c + 1 -> c[xs]
$006             jsb fst1
```

To repair this will mean an instruction will have to added in ROM 0 so that it appears before the select ROM 6 instruction.

```
$000   pwo1:     jsb pwo2
$001              no operation
$002   tms5:     select rom 6
$003   tms2:     jsb frac
```

Now the trouble is that all `select ROM N` instructions above address $002 might reference a wrong ROM address and possibly all `select ROM 0` instructions from the other ROMs will access ROM 0 incorrectly. You can see that this one simple change can cause a coding nightmare.

In later models, a `Delayed Select ROM N` instruction was introduced. This allowed a ROM to be pre-selected, but not actually turned ON until a following `GOTO` or `Jump To Subroutine` (JSB) instruction executes. This allowed code jumps to anywhere in the selected ROM and also made code changes a lot easier.

As new models came out, new functions were added and this required more ROM space to hold the code.

The `Delayed Select ROM N` instructions work with N being 0 – 15 allowing up to 16 ROMs to be selected. The HP-67 and HP-97 ROMs for example, are setup like the other calculators in that each ROM page is made up of 256 10 bit words. Unfortunately the ROM space required to fit the code for these models required more than 16 pages. This required a slightly different addressing scheme and a Bank Switching instruction was introduced. There are 2 ROM banks available, each having 16 pages of 256 10 bit words.

The ROM address map for these calculators is as follows:

```
67-97 - Bank0 = $0000-$0FFF      16 pages
67    - Bank1 = $1000-$17FF      4 Pages available
97    - Bank1 = $1000-$1BFF      8 pages available
```

Looking at the ROM space in Bank 1, you would think that another 16 pages would be available to access. The extra ROM space was not actually required in these models and the ROM chips were not installed on the circuit board.

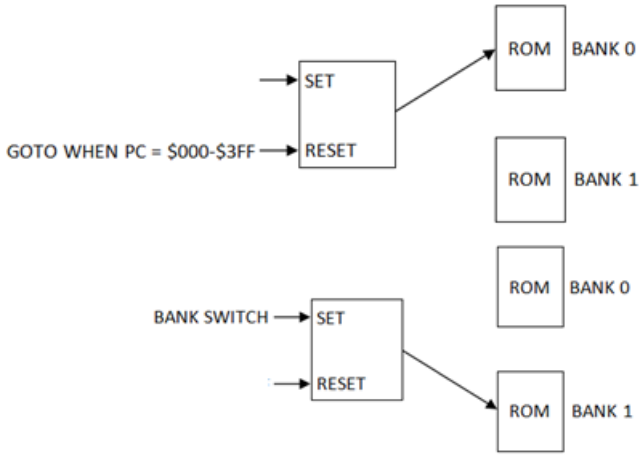
For those looking a little closer, you would expect that 8 and 12 pages would be available for the 67 and 97, however the ROM space at addresses $000 - $3FF in Bank 1 is not used as it has a special function. The `Bank Switch` instruction only swaps from Bank 0 -> Bank 1 and as a result, if you look at the code in the 67 and 97, you will see that the instruction is only used from Bank 0 from ROM addresses $7F4 to $7FE.

To allow the code to switch back to Bank 0 when required,, the hardware is setup so that if the target of a GO TO instruction address is between $000 and $3FF, the bank select bit will reset to Bank 0. If the bank select bit is already set to bank 0, then nothing changes.

These are parts of the code that executes in the HP-67 for the `[f] [-x-]` key press.

```
. . . .
$07F6        Bank Switch               << causes a Bank Switch to Bank 1
. . . .

$17F7        if no carry go to $DB     << now executing from Bank 1
. . . .

$17DB        delayed select rom 5

. . . .      << this code executes the X register display with DP flashing

$1526        b exchange c[w]
```

```
$1527         0 -> s12
$1528         delayed select rom 0
$1529         if no carry go to $F3
. . . .

$10F3         if s8 = 0                  << inside ROM Address $000 to $3FF
$10F4           then go to $054          << cause a Bank Switch to Bank 0
. . . .

$0054         CRC 1300                   << now executing from Bank 0
$0055         binary
```



A small bit of trivia:

- The microcode for the HP-67 and HP-97 is exactly the same from ROM address $400 to $FFF
- The PICK chip used in the HP-97 and HP-19C are the same

Opening a Spice Calculator

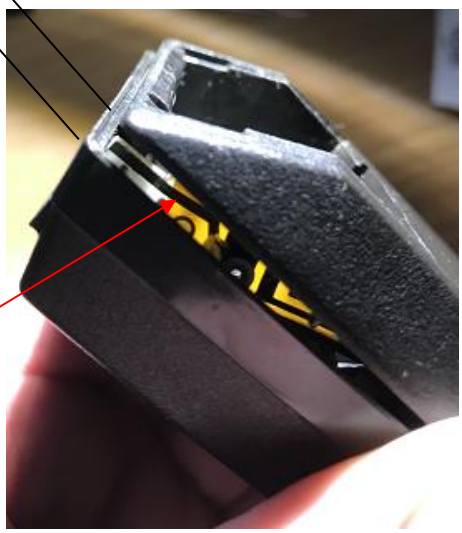
The Spice calculator case has been described as a lot more fragile with regards to construction than the other Classic models and sometimes the calculator case can be broken by trying to open it, simply because you didn't know how it was held together. This can result in excessive force being applied when the case appears to be "stuck" closed and then breakage may follow.

Some soft cloth or foam under the calculator will help prevent scratches on the case from the bench top.

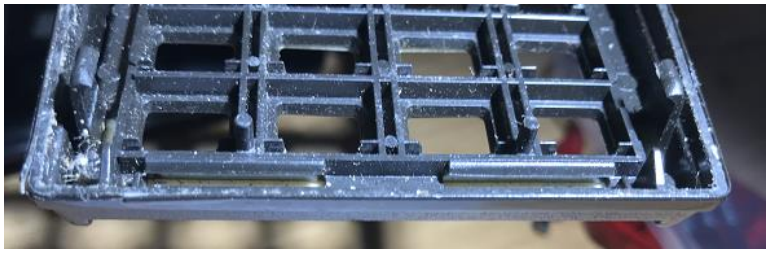
First off, remove the battery cover and batteries. There are two screws that need to be removed. When these are removed, the bottom half of the case is free to move away from the top half – but only at the top end of the calculator. **Do not** open the case halves any wider than shown in the picture or you risk damage to the other end of the bottom case.

About 1/8 inch maximum

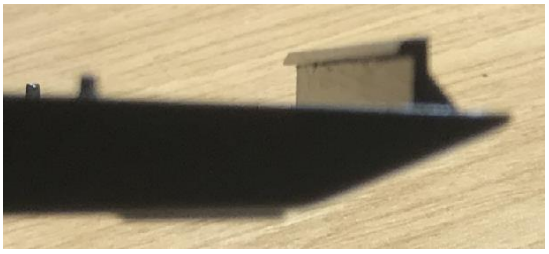
Holding the two case halves apart as shown, also frees the bottom case from the battery terminal circuit board.



The lower end of the case is where the trouble lies when trying to open it. There are two wide plastic mouldings in the top case that clip underneath lugs located in the bottom half which make the case difficult to split apart.

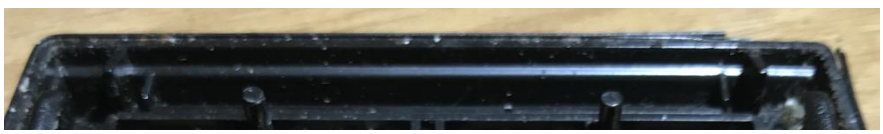


Top Half



Bottom Half

To compound the problem, the edges of the top case have a raised lip that fits snugly inside the lower case when assembled. This gives the edge all around the join a nice even finish. It would be nice to just slide the bottom case backwards in relation to the top case to disengage the locking latch, but the raised edge lip prevents this.



You should be able to see why, after removing the two screws, splitting the top case edges apart too much can break the locking latch. If this happens, glue may not fix the problem due to the forces applied to these components.

HP had a special tool for opening the cases PN: T190583 which was inserted into a groove inside the battery compartment. The shape of this tool is not known by the author.

Another way to open it is that you can slide some thin plastic along the case edges and slide it around the lower corners as shown. I used some of the same plastic box as was used for the other Classic tools described earlier as it is dense but also soft and not likely to cause damage. This will lift most of the bottom case half above the lip edge.

Using the left hand, hold the calculator keys side down with the bottom edge of calculator facing away from you. With the right hand, split open the top case edge as shown previously and grip the bottom case half somewhere near the middle of the calculator where it is strongest and apply some force towards the bottom end. Gently rotate the bottom case left (or right) to unhook the internal clip on one side then rotate the opposite direction to unhook the other side. Be firm but gentle and it should come apart.

While doing this, be mindful of the battery connect circuit board so as not the stress it.

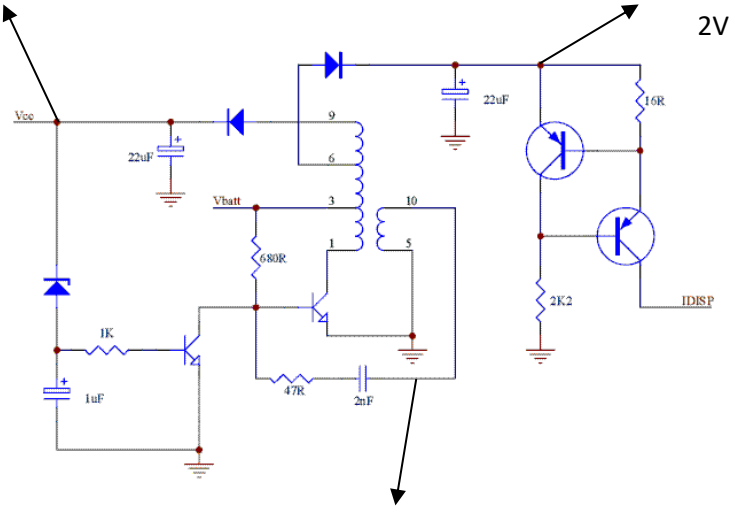
Please note that if you attempt to split the cases without raising the edges above the case lip, the plastic lip area might fracture and break off.

When reassembling the cases, make sure the battery board is properly in place and move the two halves together. Light pressure will be required on the bottom case end while also pushing forward to engage the locking latch. Make sure it all looks ok then insert the two screws. Do not overtighten them as they may strip out of the posts, or the posts themselves may split apart.

Spice Power Supply

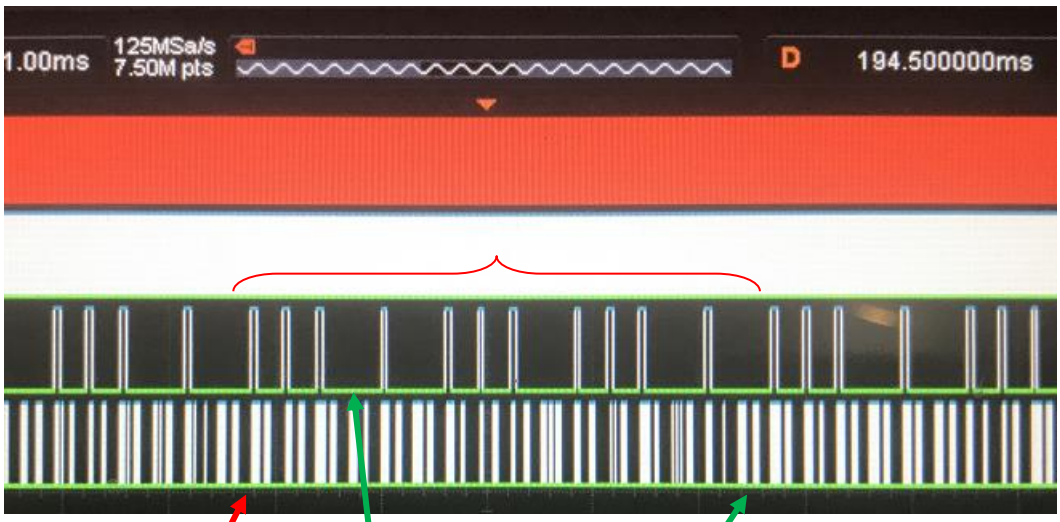
2V / 20mS per Div

2V / 20mS per Div



2V / 10uS per Div

The following image shows the key wait code loop for the HP67.



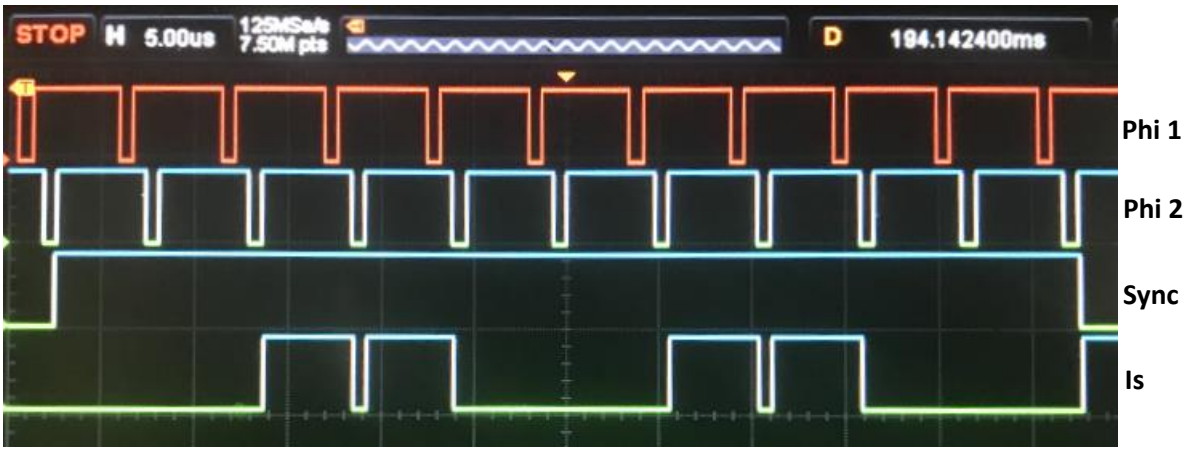

```
Addr Code   Instruction
$0086 0CC     0 -> s3                    Sync = 1
$0087 170     CRC 560                    Sync = 1
$0088 0D4     if s3 = 1                  Sync = 1
$0089 2D4       then go to $2D4          Sync = 0
$008A 3DC     if s15 = 0                 Sync = 1
$008B 077       then go to $077          Sync = 0
$0077 0CC     0 -> s3                    Sync = 1
$0078 340     CRC 1500                   Sync = 1
$0079 0D4     if s3 = 1                  Sync = 1
$007A 0B3       then go to $0B3          Sync = 0
$007B 04C     0 -> s1                    Sync = 1
$007C 0C0     CRC 300                    Sync = 1
$007D 0D4     if s3 = 1                  Sync = 1
$007E 084       then go to $084          Sync = 0
$007F 2DC     if s11 = 0                 Sync = 1
$0080 086       then go to $086          Sync = 0
```

You can see that there are missing pulses in the Sync Bus. When the Sync pulse is present, the data fetched from ROM is used for the next program instruction. When the Sync pulse is missing during these clock cycles, the data from ROM is used for the next Program Address.

The Sync pulse is held low after an **`IF`** type of instruction is executed – such as **`if S3 = 1`**. The next instruction that is fetched (regardless of what it may be) is then considered to be a ROM address and the instruction becomes an *Implied Goto*. By doing things this way the full 10 bits of the fetched ROM data can be used to *Goto* anywhere in the 1K memory space of the active ROM chip. Normal instruction formats cannot do this because 2 of the 10 bits are be used for the actual instruction decoding and therefore 8 bits are available for the new ROM address. This limits the jump to the current ROM page $xx00 to $xxFF.

Based on this information, you should be able to determine which instruction is which in the above image.

This is an expanded view of the Sync pulse which occupies the last 10 cycles of the 56 Phi1 and Phi2 clock cycles used for each instruction.

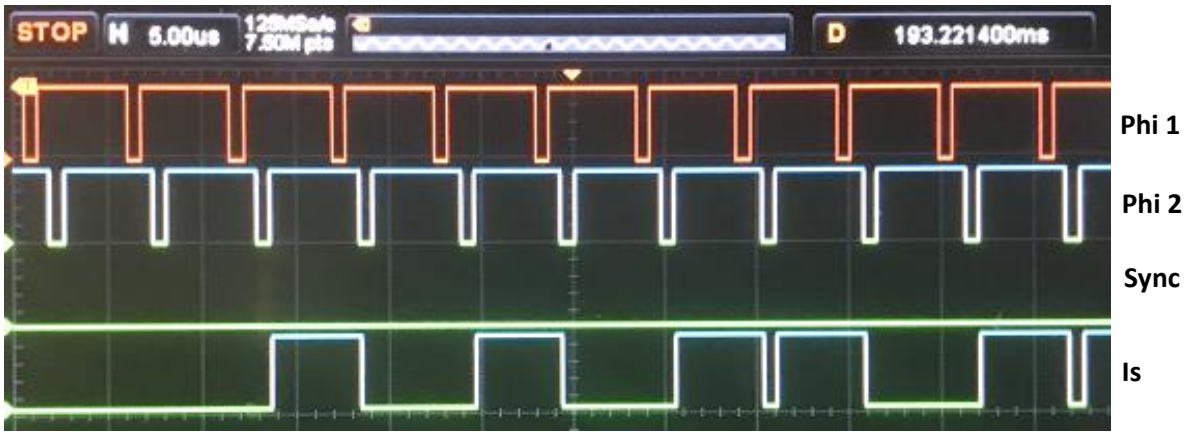


Bits	0	1	2	3	4	5	6	7	8	9	
Data	0	0	1	1	0	0	1	1	0	0	
Cycle	46	47	48	49	50	51	52	53	54	55	0

The low bit of the instruction is sent out on the Is Bus first so the 10 bit data becomes an instruction:

```
00 1100 1100     $0C          0 -> s3
```

The next image shows the data on the Is Bus when the Sync pulse stays low.

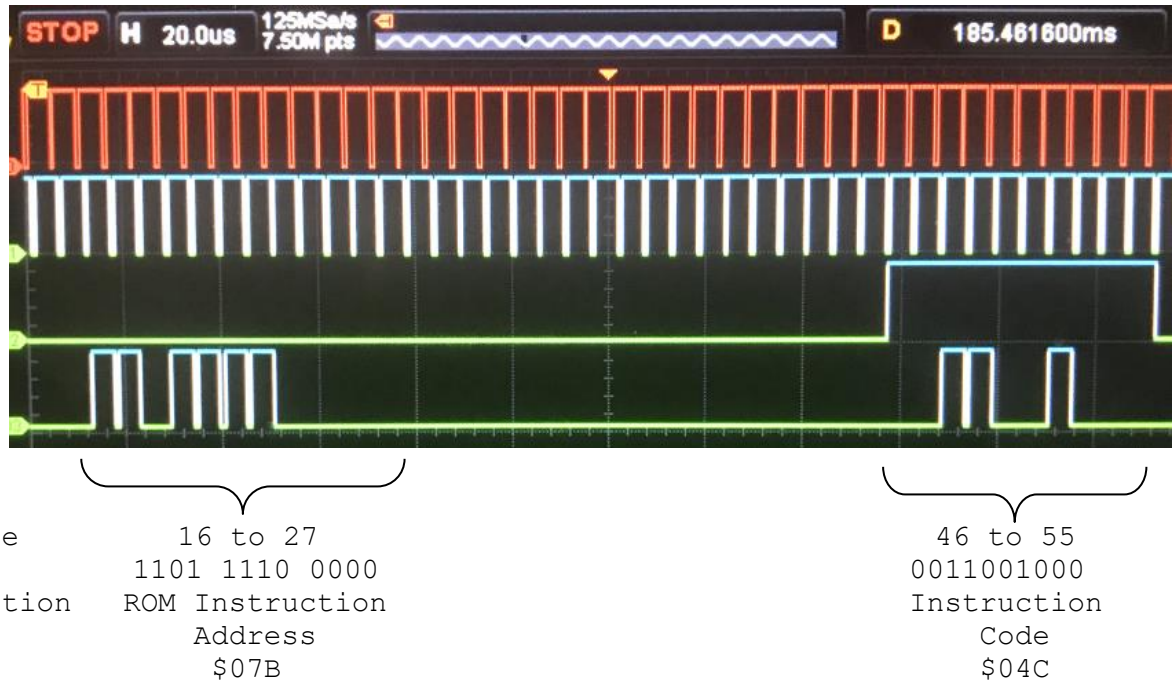


Bits	0	1	2	3	4	5	6	7	8	9	
Data	0	0	1	0	1	0	1	1	0	1	
Cycle	46	47	48	49	50	51	52	53	54	55	0

The low bit of the instruction is sent out on the Is Bus first so the 10 bit data becomes a ROM address:

```
10 1101 0100     $2D4         then go to $2D4
```

This image shows how the instruction pipelining works.



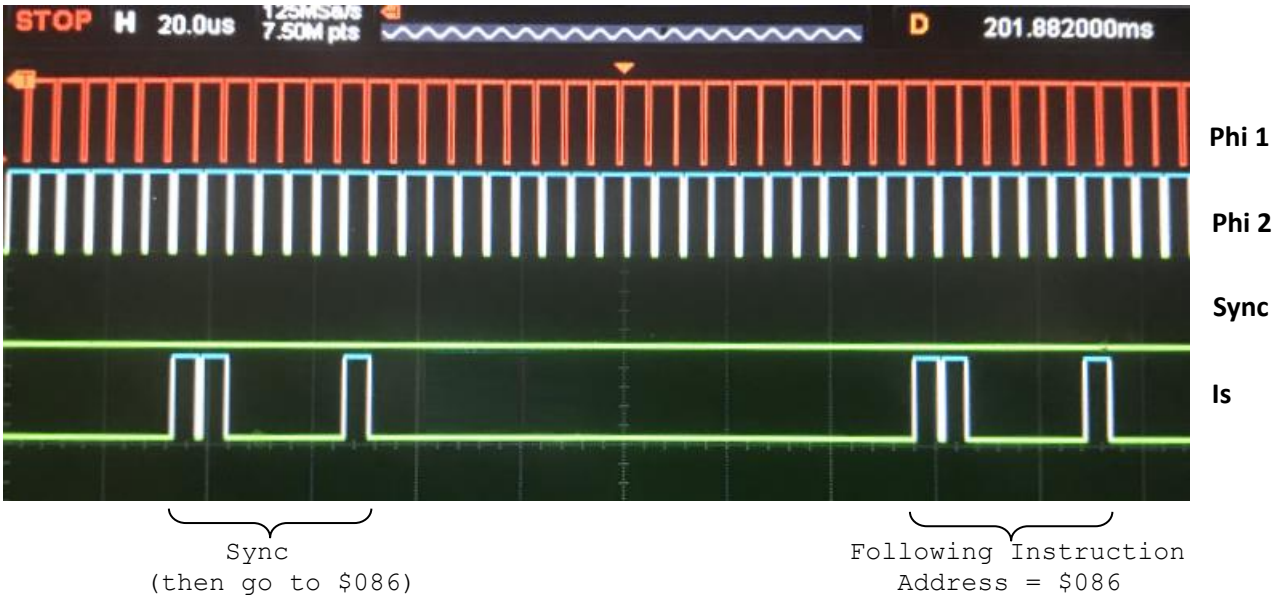

```
Cycle          16 to 27                    46 to 55
             1101 1110 0000              0011001000
Function   ROM Instruction               Instruction
               Address                      Code
                $07B                        $04C
```

The ROM instruction address is 12 bits wide LSB first. Once this is sent to the ROM during Phi1/2 bit times 16 to 27, the following 19 cycles gives time for the ROM to read its ROM array and start outputting the data at bit times 46 to 55 LBS first. This instruction is executed during the following 56 bit instruction time.

The above information is for the following instruction:

```
Addr Code  Instruction
$07B  04C   0 -> s1                          Sync = 1
```

The next image shows a missing Sync pulse which sets the ROM address, and then the following ROM address matches that address and is where the next instruction will be fetched.



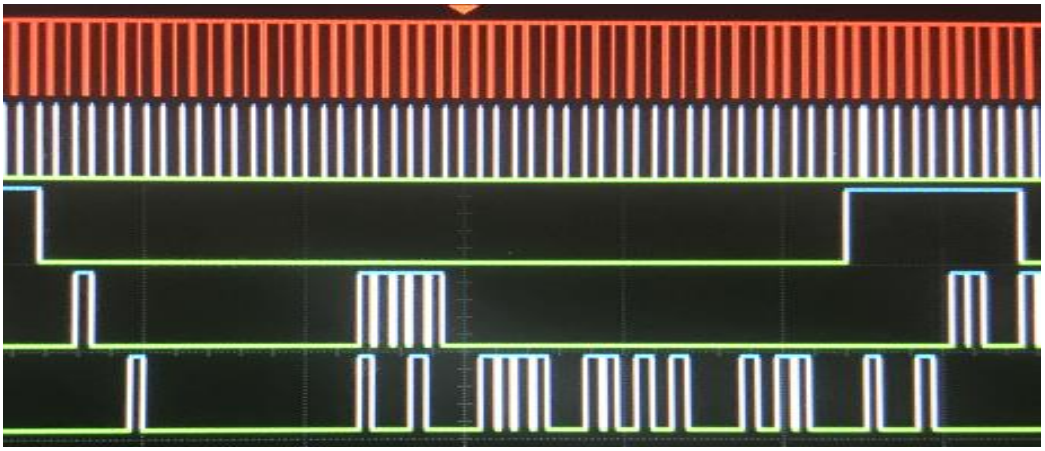

```
        Sync                    Following Instruction
(then go to $086)                 Address = $086
```

The next image shows the calculator in the key wait loop but with 1 2 3 4 5 6 7 8 9 . on the display.

As numbers are entered into the calculator, the information is placed into the C register and appears on the Data line. The C register will contain the following data

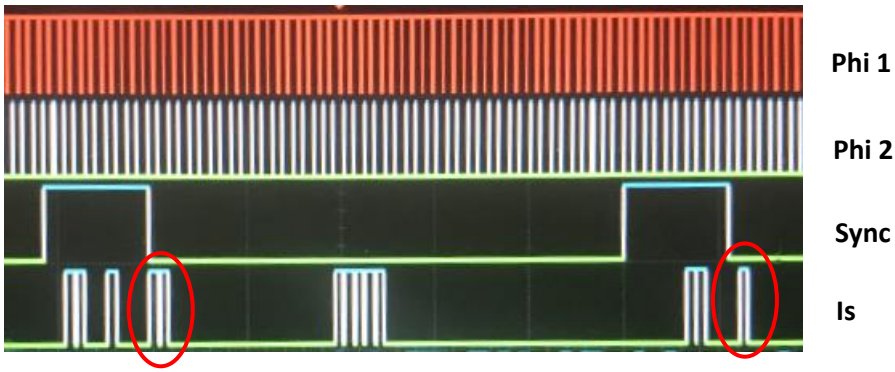
```
S M M M M M M M M M S E E      S = Sign   M = Mantissa     E = Exponent
0 1 2 3 4 5 6 7 8 9 0 0 0 8
```

The data is represented by 56 bits of information starting at the Exponent LSB first.



8 0 0 0 9 8 7 6 5 4 3 2 1 0

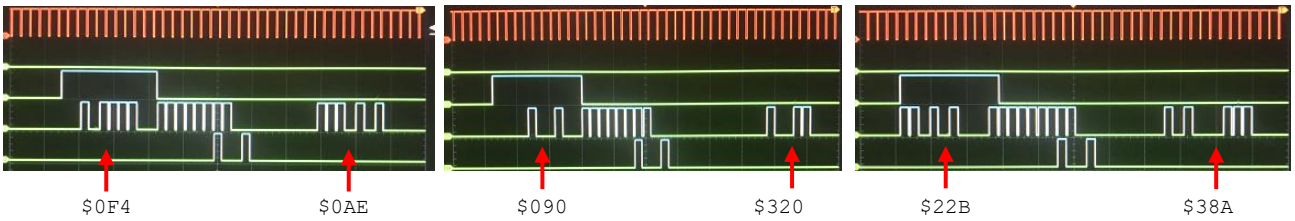
The next image shows the display data that is decoded by the ROM0 IC. This chip has 1K of ROM, no RAM, and also decodes the display information and sends it to the anode driver on every instruction cycle. Bits 0 – 3 of the 56 bit instruction cycle displays any of 16 digits which can be 0 1 2 3 4 5 6 7 8 9 r C o d E [ ]. This gives the display the ability to display all numbers and “Crd” and “Error”.



Digits 3 2

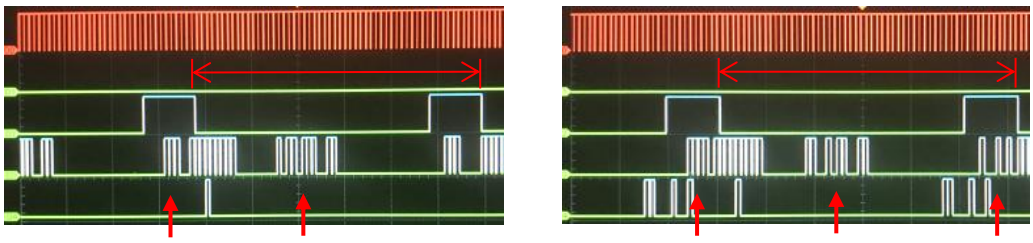
Bits 4 – 7 are supposed to be for Decimal Point, Sign and blanking but it was not conclusive to determine what was what. Perhaps later ☺

These images show the following instructions execute when a `Delayed select ROM` instruction executes.



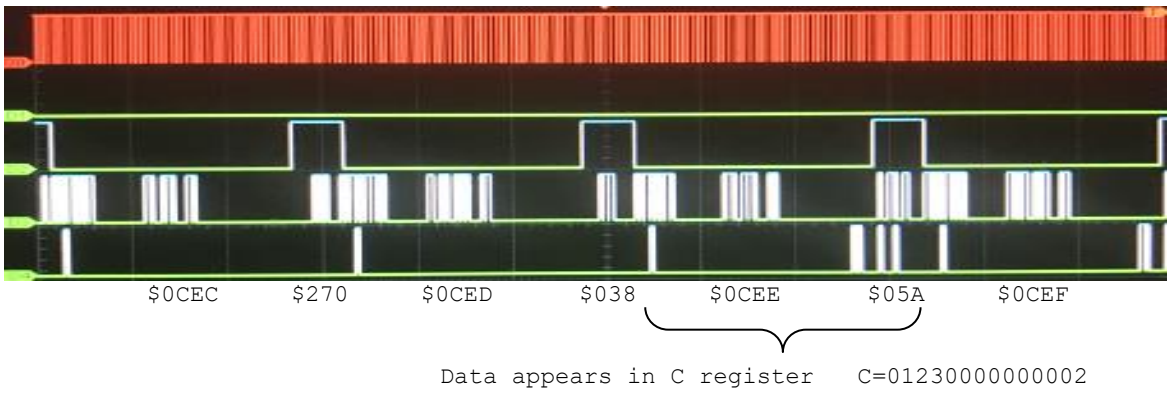

```
$0AD 0F4      delayed select rom 3
$0AE 090      a -> rom address
$320 22B      if no carry go to $8A
```

These images show after 123.00 was entered stored to RAM register 1 using [STO] [1]



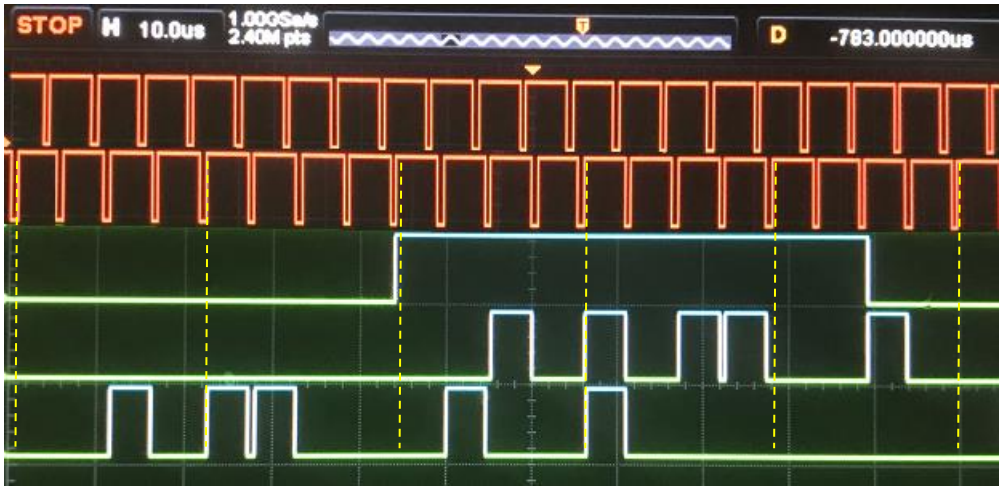

```
$CEC 270      c -> data address   C=00000000000001    ROM accepts RAM address 1
. . . .
Code recovers C register ready for storage
. . . .
$CD2 2F0      c -> data           C=01230000000002    ROM accepts C into RAM 1
$CD3 148      m1 -> c
```

This image shows the instructions that read the RAM in the ROM chip after [RCL] [1] is pressed



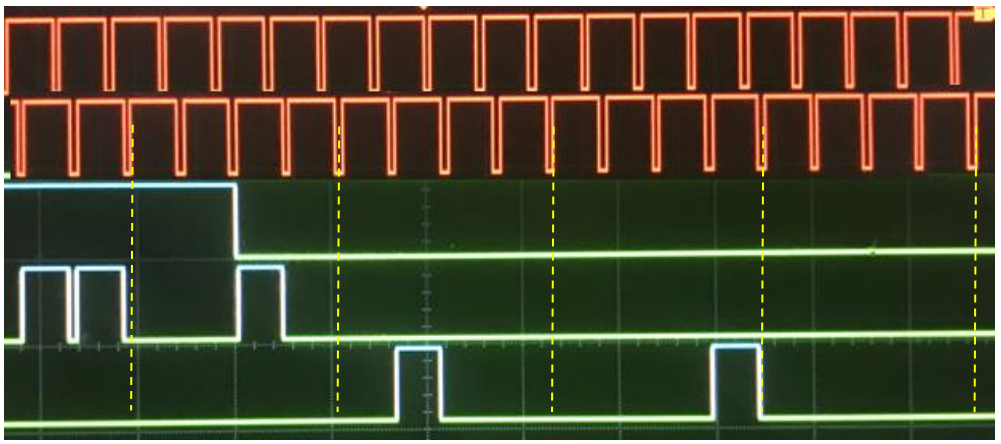

```
$CEC 270      c -> data address          C=00000000000001
$CED 038      data register 0 -> c       C=01230000000002
$CEE 05A      a exchange b[w]
$CEF 09A      a exchange c[w]
```

This image shows the first 5 bits on the data line when C = 01234560000082 (123456 E88 ENTER)
You can see that the fist data bit is on bit 2 of the instruction cycle



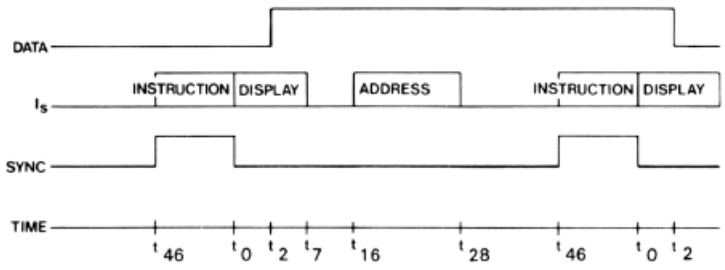
0 0 1 0 1 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0
4 3 2 1 0

This image shows a continuation from the right hand side of the above image.

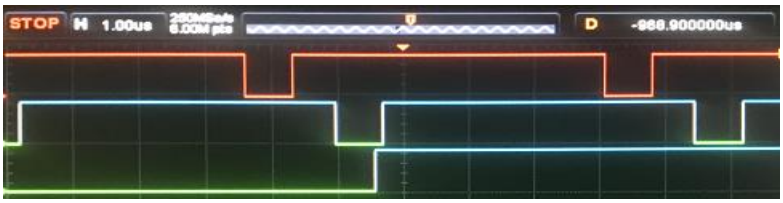


1 0 2 8 0

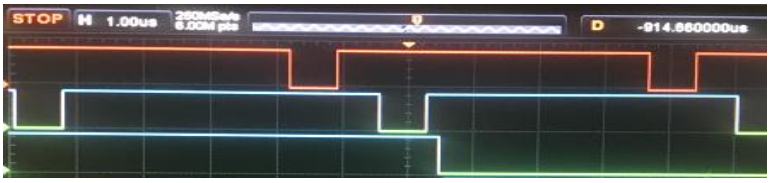
This matches with the timing diagram shown in the HP-97 service manual.



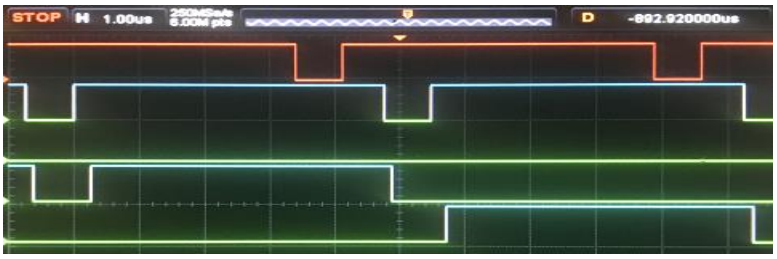
Detail: Relationship between Phi1 Phi2 and rising edge of sync



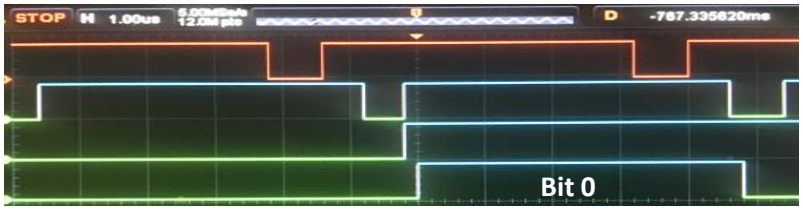
Detail: Relationship between Phi1 Phi2 and falling edge of sync



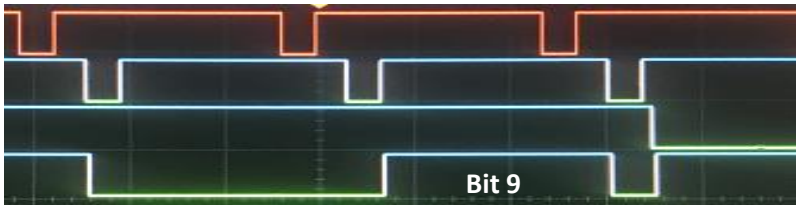
Detail: Relationship between Phi1 Phi2 and rising / falling Is and Data



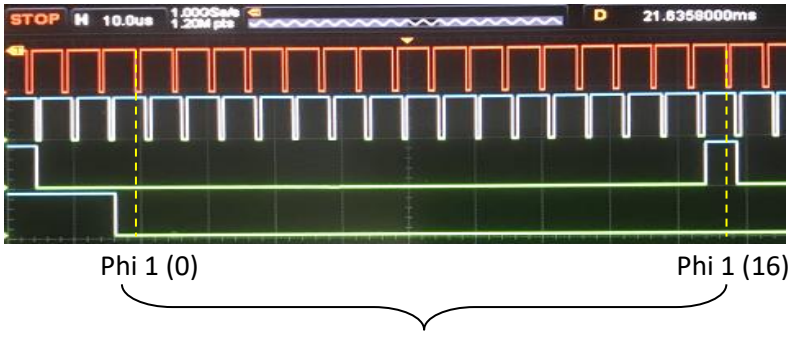
Detail: Relationship between Phi1 Phi2 and rising Sync and bit 0 of instruction – Bit 0 = Logic 1



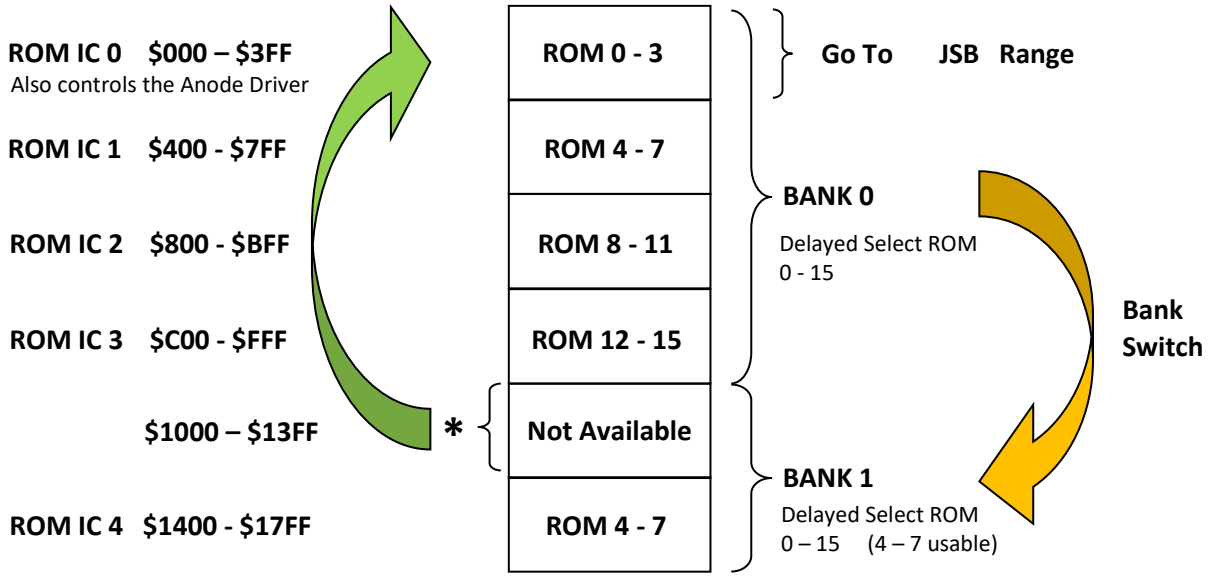
Detail: Relationship between Phi1 Phi2 and falling Sync and bit 9 of instruction – Bit 9 = Logic 1



Detail: Relationship between SYNC = LO and bit 0 of address – Bit 0 = Logic 1



The HP-67 ROM map is shown below.



Each instruction code is only 10 bits wide so they cannot directly access every address in ROM. `Go To` and `JSB (Jump To Subroutine)` can only access a single ROM which occupies 256 words. To execute code in other ROMs the Delayed Select ROM instruction can be used. This instruction pre-sets the next ROM to use (0 – 15) and if the next instruction is `Go To` or `JSB`, control is passed to that ROM.

As there are only 12 ROM address bits available, the ACT can only access ROM addresses $000 to $FFF. However there are more ROMs available and a larger address space is required. Therefore the ROM address space is split into **Bank 0** and **Bank 1**. To access the ROMs in **Bank 1**, the `Bank Switch` instruction is used.

All ROM's *listen* out for this instruction and they react in one of two ways - either listen on the **Is** line for a valid 12 bits ROM address or not. For the HP-67, only one ROM IC is programmed to listen in this way and that is ROM 4, all the others stop listening. Note: The HP-97 has 2 of these ROMs. The *Bank Switch* instruction does not affect the Program Counter (ROM Address), it will increment by one as normal after the instruction executes and the ACT treats the instruction as a `No Operation`.

Example:

```
$0D3 1F4   delayed select rom 7   ROM IC 0 - ROM 0 in Bank 0 is active
$0D4 3D1   jsb $F4                ROM IC 0 - ROM 0 in Bank 0 is active
. . . .
$7F4 230   bank switch            ROM IC 1 - ROM 7 in Bank 0 is active
$7F5 11A   0 -> c[w]              ROM IC 4 - ROM 7 in Bank 1 is active
. . . .
$495 210   return **              ROM IC 4 - ROM 7 in Bank 1 is active
$0D5 0B4   delayed select rom 2   ROM IC 0 - ROM 0 in Bank 0 is active
```

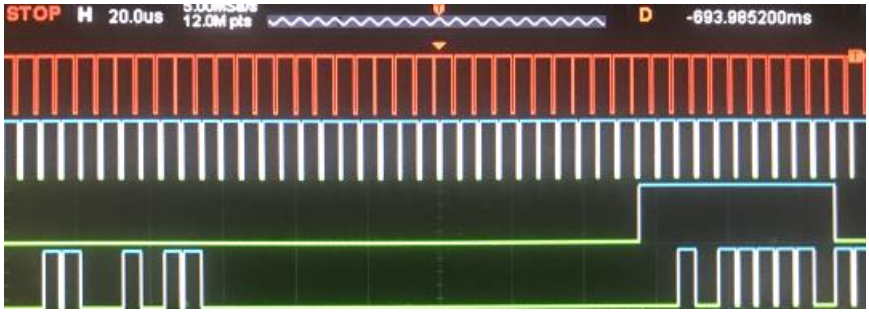
For this system to work in the HP-67, the Bank Switch instruction can only be in `ROM IC 1, ROM's 4 – 7`. If you look at the HP-67 microcode listing you will see all 11 of the `Bank Switch` instructions are in ROM 7 starting at address $F4.

***** The `Bank Switch` instruction can only activate ROM IC 4 in **Bank 1**. (*The HP-97 has 2 of these ROM's*) It cannot be used to switch back to **Bank 0**. To accomplish this task, the programmer writes code that selects any ROM address space in the range $000 - $3FF. All ROMs react to this address range and the ones that are programmed to activate on the `Bank Switch` instruction stop listening on the **Is** line and ROM 0 starts listening. There is no ROM 0 available in Bank 1 as this ROM space is used solely to switch back to Bank 0.

****** Notice the `Return` instruction in the above code executed in **Bank 1**. This caused the Program Counter to reset to $0D5 (`jsb $F4` + 1) which is in the ROM 0 space, so **Bank 1** cancels and **Bank 0** becomes active.

These images show the program flow for the above code.

ROM IC 0 – ROM 0 in Bank 0 is active



Phi 1

Phi 2

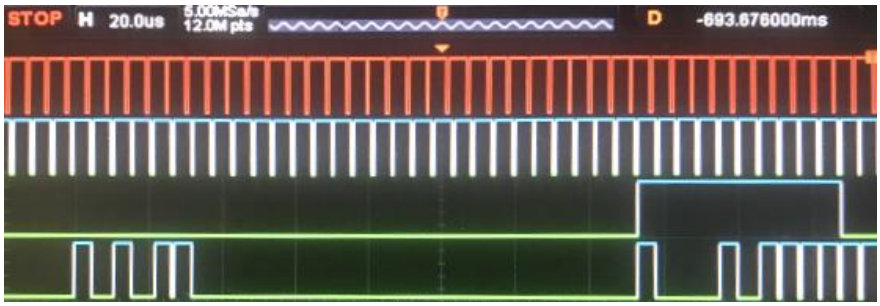
Sync

Is

$0D3

$1F4
delayed select rom7

ROM IC 0 – ROM 0 in Bank 0 is active



Phi 1

Phi 2

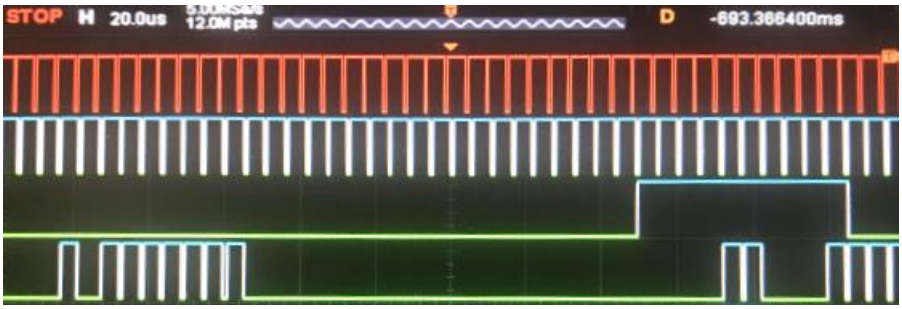
Sync

Is

$0D4

$3D1
jsb $00F4

ROM IC 1 – ROM 7 in Bank 0 is active



Phi 1

Phi 2

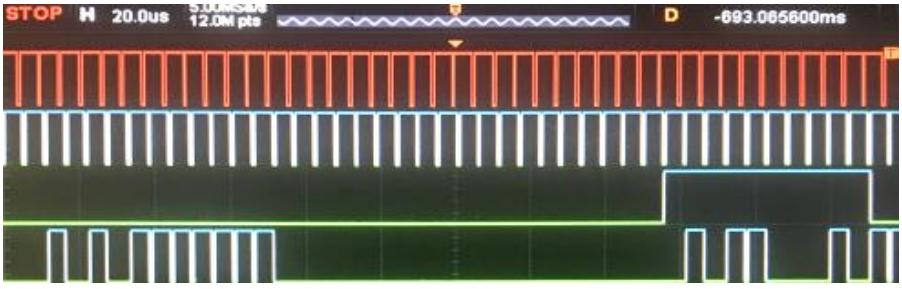
Sync

Is

$7F4

$230
bank switch

ROM IC 4 – ROM 7 in Bank 1 is active



Phi 1

Phi 2

Sync

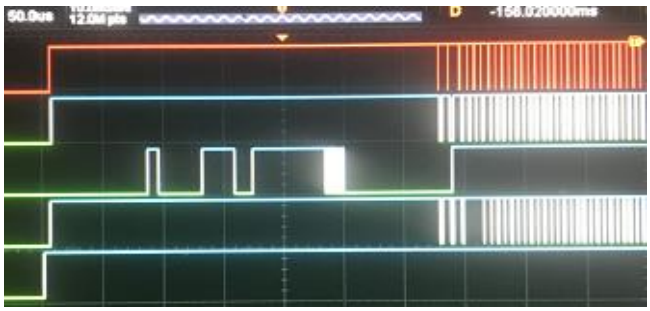
Is

$7F5
Effectively $17F5

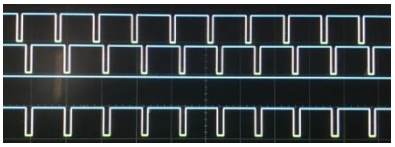
$11A
0 -> c[w]

HP-67 Power On

About 330uS after switching on, the signals start stabilizing.

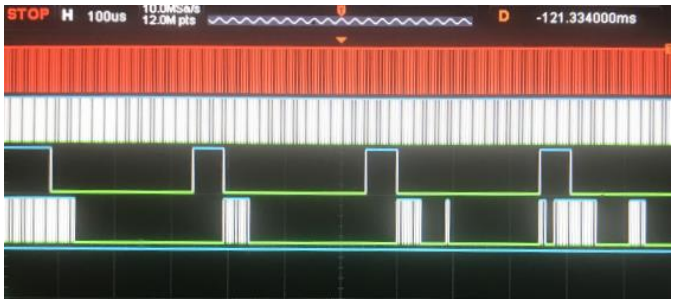


About 35mS after switching on, the SYNC becomes active and instructions begin to get fetched from ROM.



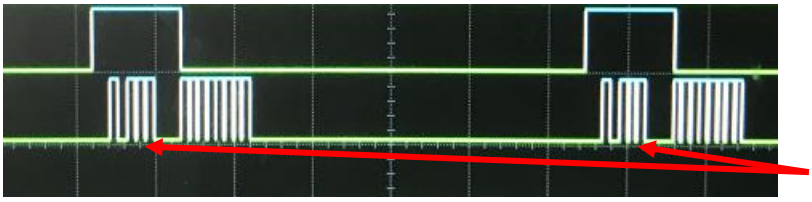
You can see that the **Is** bus follows the **Phi 2** signal in reset while the SYNC line is high

The 1st valid SYNC pulse seems to be ignored as the ROM address does not increment to one until the 3rd.



The HP-97 power up shows more clearly that the 1st 56 bit sequence is ignored and the 2nd is used for the first instruction.

1st SYNC Pulse



2nd SYNC Pulse

Same instruction

ROM Addr $000, $074, delayed select rom 1

This might happen because the DATA bus starts on bit cycle #2 and the first 56 bits will not have all the required information, although it doesn't appear that the DATA bus is accessed during these times.

The serial buses like **Is** and **Data** have multiple IC's connected to them and those IC's can take control of it when data needs to be transmitted. To avoid bus data conflicts and possible short circuits when the bus is in the opposite logic state, only one IC will control the bus at any one time and can only pull the bus to one logic state.
With the **Is** bus disconnected from the ACT, normal bus signals appear from the ROM IC's. If the **Is** bus is tied to Vcc (+6V) via a resistor, the bus signals remain HI. If the **Is** bus is tied to ground via a resistor, the bus shows HI and LO states but the HI state has a slightly reduced amplitude. This seems to indicate that the **Is** bus is loosely pulled LO by a circuit path inside the IC's and it is only directly pulled HI by the active IC, but never directly LO otherwise a short circuit condition could exist.
Reading from Memory

The next image is what happen when `Load Constant 3` is executed when the P register = 1, this sets up the RAM address to equal $30.

3 appears in bits 5 and 6 (Exp Tens) and also has other data from previous `Load Constant 3 (0D8)`

The next image shows what happens with a `data register 14 -> C` occurs. The computed RAM address is $30 + 14dec = $3D. This register stores the display formatting data which is set by the [DSP] key among others. This register has the default power up value of [00000012220000] stored in it which is FIX 2 DEG mode.

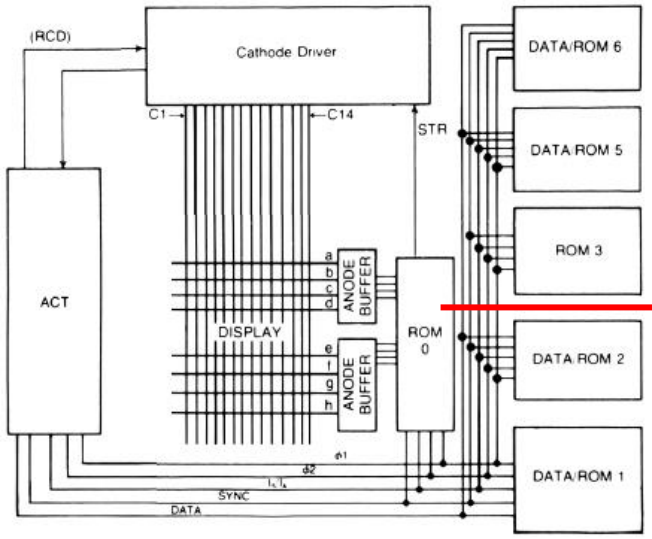
The `data register 14 -> C` instruction has come from the ROM address $040B, while the SYNC signal is HI. On the following 56 bit word time, the Program Counter is incremented to $04C0 and the instruction executes immediately when the SYNC signal goes LO. The first two clock cycles give the ROM time to process the instruction and then the 56 bits contents of RAM register $3D is then output onto the DATA bus, LSB first starting at clock cycle 2. The 54th and 55th data bits appear on the DATA bus during clock cycles 0 and 1 of the following instruction cycle.

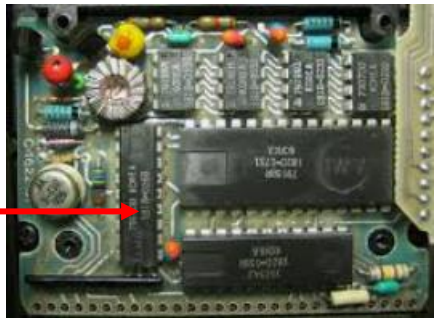
0 0 0 0 2 2 2 1 0 etc

`data register 14 -> C` ROM Address = $40C

ROM 0

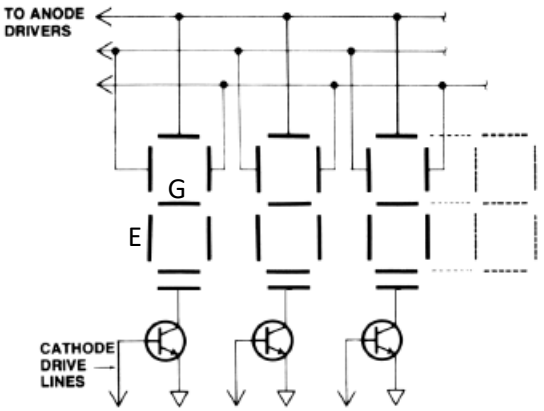
The ROM 0 chip holds 1K of program words like the other 8 pin ROMs in the Woodstock calculators, but it also decodes information on the **Is** bus to drive the LED anodes of the display.

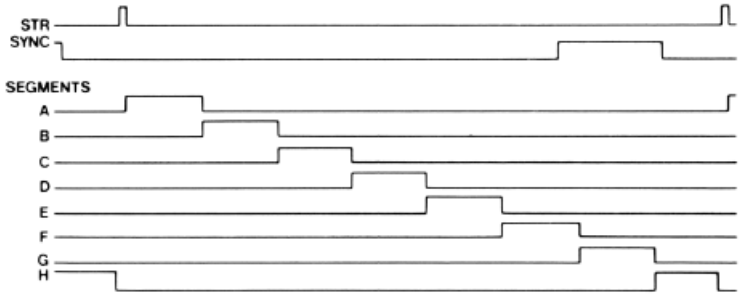




There are 15 LED displays in these calculator models, but there are only 14 drivers for them.

The sign digits for the mantissa and the exponent share LED digit 3 from the cathode driver. When the cathode driver turns on digit 3, segment E is energised when the Mantissa sign is negative, and segment G is energised when the Exponent sign is negative. When segment E is energised for digit 3, it would not normally show a correct negative sign because segment G has to be lit. There are some transistor switches connected to these segments which route the signals to the correct LEDs at the proper time. The other 7 LEDs in the sign digits are not used and always stay off. In the Topcat series, such as the HP-97, again only the G segments are required for the sign digits and these only have that segment physically in the displays. To cut costs, there's no point including them if they have no use.

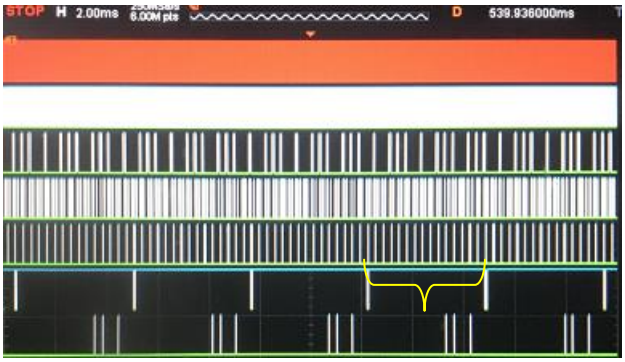




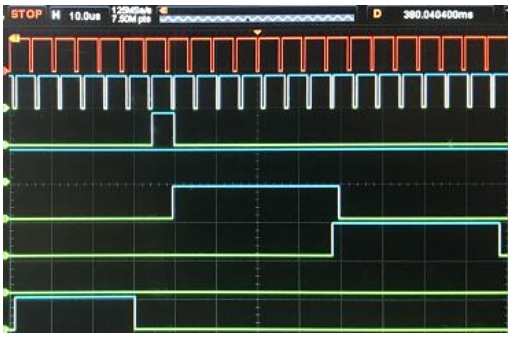
The cathode driver receives 2 input signals, Reset Cathode Driver (RCD) from the ACT, and Strobe (STR) from ROM 0. The STR signal tells the cathode driver to turn each digit on in turn such that only one digit is on at any one time. The RCD signal is used to reset the cathode driver back to digit 1.

ROM 0 uses the SYNC pulse to know when to issue the STR pulse and it is also responsible for outputting the digit anode data on the **Is** bus at the correct time. The ACT is responsible for issuing the RCD pulse. These signals make sure each digit shows the correct information when turned on. You can see from the diagram above that only one LED is turned on at any one time. This keeps the overall current use down, and the refreshing takes place so fast that the human eye cannot perceive any flickering. As the decimal point is only a small dot, the time that it is one is about 2/3 the time of the other segments otherwise it would appear too bright.

This scope image shows the 15 STR pulses in relation to the RCD pulse. Below that you can see segment A of the LED digits lit up. This display is showing **0.00** and the first mantissa digit is the last digit output to the display before the RCD pulse occurs.



15 STR pulses



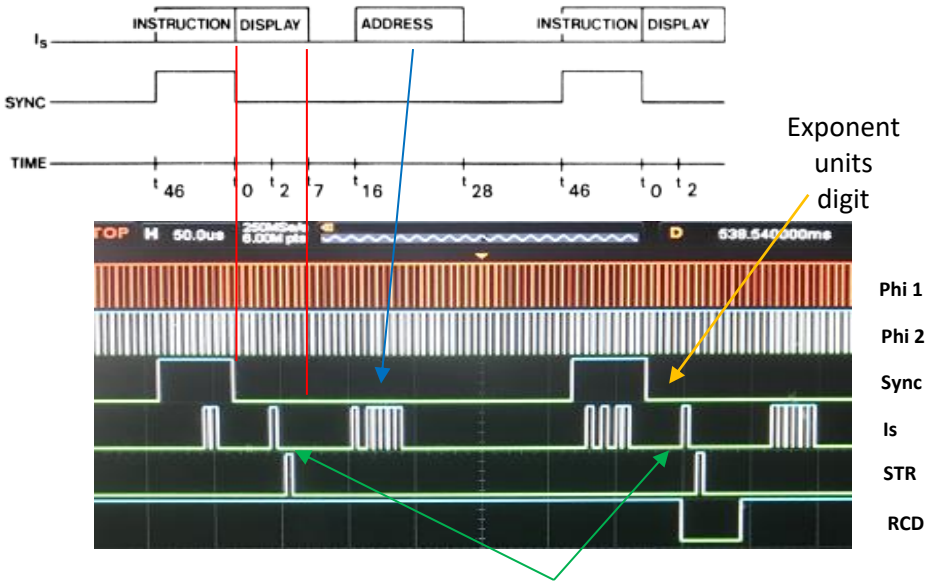
This shows the relationship between the STR pulse and the start of segment A and how there is a small overlap between segments lighting up.

You can also see the DP LED is on for about 30uS compared to 40uS for the other LEDs

There is about a 5uS gap between the end of the DP LED and the start of the 5uS STR pulse

The 56 bits that make up the instruction cycle take 320uS. For the single LED digit updates, there are 7 segments at 40uS, 1 DP at 30uS + 5uS gap and 5uS STR pulse. The complete display refresh takes 4.8mS and for those good at maths, this equates to 15 digits, not the 14 mentioned previously. This is because there are actually 15 STB pulses per display refresh.

The data for the ROM 0 IC is output on the **Is** bus during bit times 0 to 7.

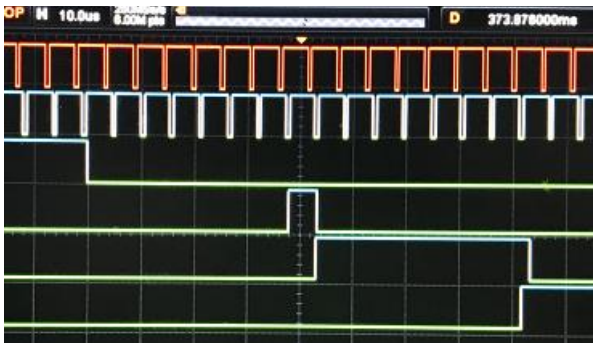


This data for the display is `00000100` (LSB first). Data = $20 = Blank. If you look closely, you can see that the next display data is the same $20 when the RCD pulse occurs. These correspond to the 15th and 1st STB pulses. The display data for the 15th pulse always seems to be the same as the 1st.

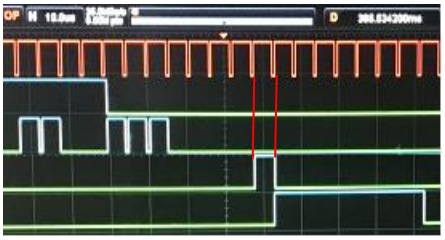
You can also see how the RCD pulse overlaps with the last STR pulse. The first digit out is the exponent units, followed by exponent tens, the signs digit, the mantissa digits 11 down to 1. I assume from this that the 15th display data is discarded.

From these signals it appears that the cathode driver resets on the low going edge of the RCD pulse.

This scope image shows the relationship between the edge of the SYNC line and the start of the STR pulse and Seg A and Seg B start.

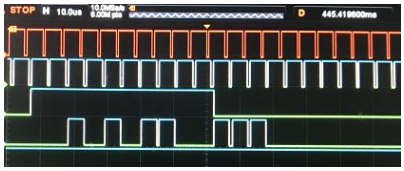


This shows digit 7 being output.
You can see that the STR pulse occurs on bit 7 of the display data. Segment A is lit on the LO going edge of the STR pulse

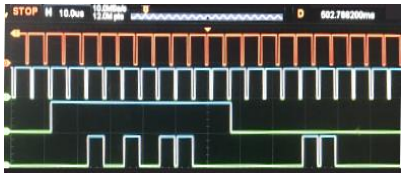


The digits appear to be decoded as in the following list.

Binary	HEX	Digit
00000000	$00	0
00000001	$01	1
00000010	$02	2
00000011	$03	3
00000100	$04	4
00000101	$05	5
00000110	$06	6
00000111	$07	7
00001000	$08	8
00001001	$09	9
00001010	$0A	o
00001011	$0B	C
00001100	$0C	r
00001101	$0D	d
00001110	$0E	E
00001111	$0F	Blank
00110000	$30	DP
0100xxxx	$4x	Blank



For displaying “Error” and “Crd”



Digit Out Order

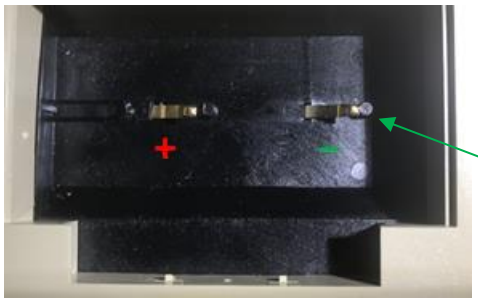
1 Exp units
2 Exp tens
3 Mant Exp Signs
4 Mant Digit 11
5 Mant Digit 10
6 Mant Digit 9
7 Mant Digit 8
8 Mant Digit 7
9 Mant Digit 6
10 Mant Digit 5
11 Mant Digit 4
12 Mant Digit 3
13 Mant Digit 2
14 Mant Digit 1
15 Exp unit duplicate

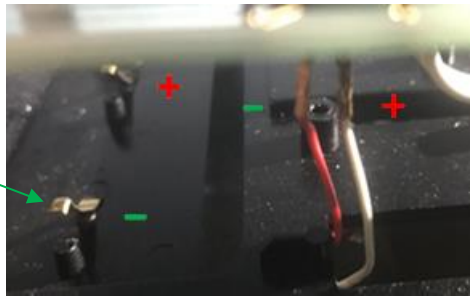
The sign digit is also decoded inside the ROM 0 IC. It uses the first 2 bits of the 8 bit display data and is only used for digit position 3.

Bit 0	Bit 1	Mantissa	Exponent	g LED	Mantissa	Exponent
0	0	-ve	+ve		On	Off
0	1	-ve	-ve		On	On
1	0	+ve	+ve		Off	Off
1	1	+ve	-ve		Off	On

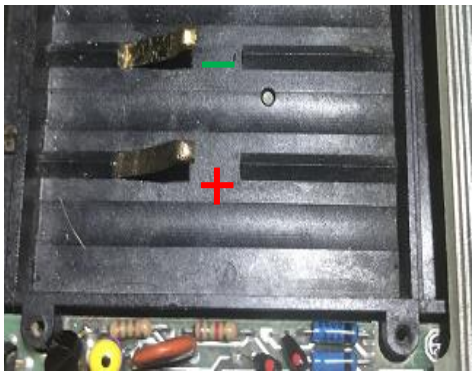
Batteries

Topcat
5V

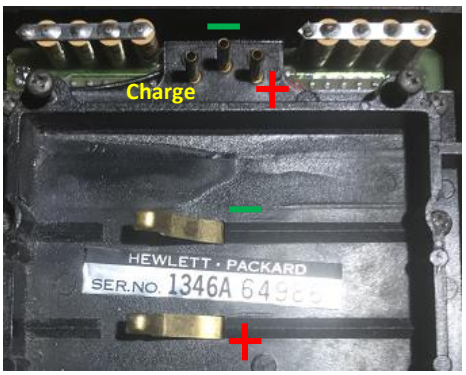




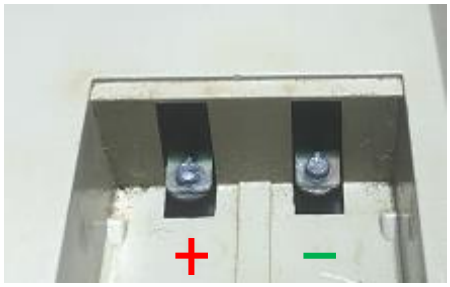
HP-67
3.75V



Classics 3.75V



Woodstock
2.5V

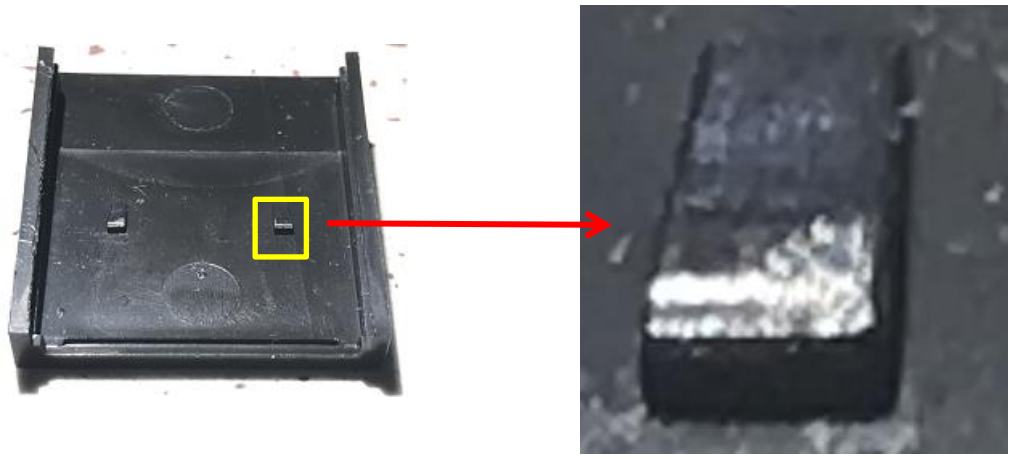


Spice
2.5V

HP-82104

Sometimes a repair of the HP-41 card reader is required. After disassembly, the repair might require the removal of the cover that surrounds the HP-41 port connection pins.

It is not immediately apparent how this cover is held in place, but it simply slides on and is held in place by two moulded locking tabs located underneath the cover. These allow the cover to slide on, but not off.

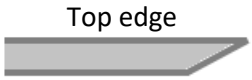


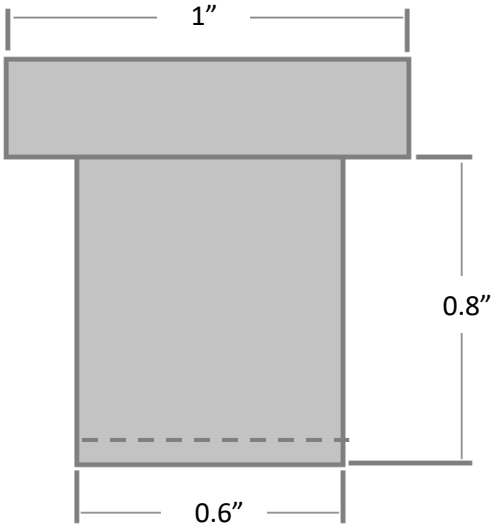
An easy way to remove the cover without damaging the cover or the connecting pins is as follows.

I used some plastic cut to size from one of those cheap disposable food containers. It is rigid enough for the job and just thick enough. Metal could also be used if it was the same thickness and all rough edges removed.

The dimensions of the tool are shown in the drawing

.

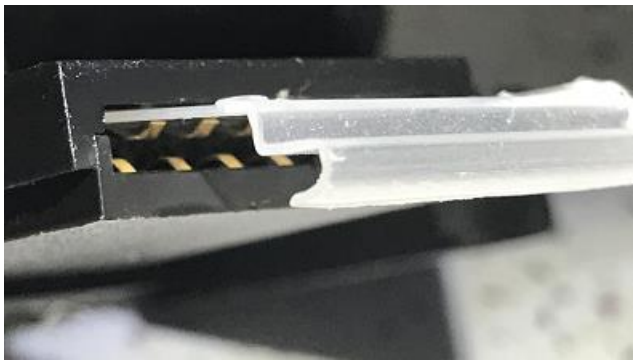




The sides of the container have a lip around the top edge, this can provide stiffness for the top edge of the tool.

Once cut out from the plastic container and trimmed to size, use a small file to chamfer the bottom edge. This will make it easier to slide the tool into the reader.

Slide the tool into the reader, underneath the top edge and above the top row of the connector pins as shown, with the tool top edge uppermost.



As you push it in, it will stop on the inner edge of the casing. Push it in a little harder and it should go in another ¼" or so raising the cover locking pins above the inside casing. Now just simply slide the top cover off away from the card reader casing.

Printer Codes

The printer codes are assembled by the HP microcode and end up in the C register. The C register can be sent to the RAM memory or recalled from RAM memory as in a STO and RCL key presses. This is accomplished in code with instructions like this.

Recall This register is where the program step pointer is located.

```
1 -> p                C = xxxxxxxxxxxx0   (x = don't care)
load constant 3       C = xxxxxxxxxxx30
c -> data address     RAM address = 30hex
data register 13 -> c Recalls RAM memory address 3Dhex to C
```

Store This is setting the register 3Ehex with default data for DSP=DEG, FIX=2 etc.

```
0 -> C[W]             C = 00000000000000
7 -> P
load constant 1       C = 00000010000000
load constant 2       C = 00000012000000
load constant 2       C = 00000012220000
load constant 2       C = 00000012220000
no operation
c -> data register 14 Note: The RAM address was previously set to 30hex
```

IC's like the PIK and CRC are accessed as though they are memory. Data is also transferred via the C register and other instructions are used to tell these IC's whether to read from C or write to C.

These two instructions are used by the PIK IC

PIK1660	The print data is treated as 6 bit codes
PIK1720	The print data is treated as 4 bit codes

Using 4 bit codes means that more character data can be squeezed into the C register and are handy for printing long numbers, and not forgetting that the C register is made up of 56 bits assembled as 14 x 4 bit nibbles.

In the printer code listings above, the (*) symbol is used to signify that these codes can be interpreted as 4 or 6 bits per character. It makes no difference if they are sent with PIK1660 or PIK1720. However, with the PIK1720 instruction, only the upper 4 bits are sent as data. When the PIK IC interprets these 4 bits, it automatically appends two more bits sets as 1.

For example:

HP-97 Character '0"	PIK1660	PIK1720	
Sent to PIK as	03	0	
	6 bit data	4 bit data	PIK appends 11 internally
	000011	0000	000011

Print Example: User enters 4 ENTER on a HP-97 with the print mode switch set to NORM.

Firstly, he HP micro code will assemble the print data into the C register for a CRC 1660 instruction. The assembly for 6 bit instructions is a bit more difficult as each character occupies 1 ½ nibbles.

RegC = FFFFFFFBE00736 when CRC 1660 executes.

Broken down as 4 bit binay values.

```
F    F    F    F    F    F    F    B    E    0    0    7    3    6
1111 1111 1111 1111 1111 1111 1111 1011 1110 0000 0000 0111 0011 0110
```

Changed to 6 bit data fields.

```
11 111111 111111 111111 111111 111011 111000 000000 011100 110110
```

Now we can associate each 6 bit field with a print character.

```
11 111111 111111 111111 111111 111011 111000 000000 011100 110110
   3F     3F     3F     3F     3B     38     00     1C     36
   EOL    EOL    EOL    EOL    BLANK  E      N      T      UPARW
```

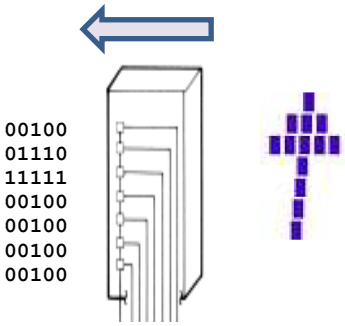
The PIK will then look up its character ROM with the character code addresses to get the print head information that will energise the print heads while the carriage moves across the screen.

As the printer prints from right to left, the first character out will be the UPARW.

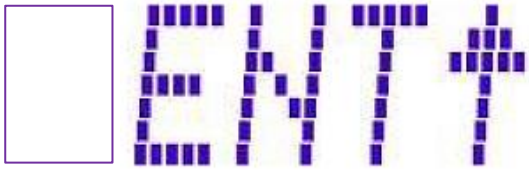
The ROM data for this character is:

```
00100 01110 11111 00100 00100 00100 00100
```

Each print head will be energised for every [1] bit in the data. There are seven pixels in the print head arranged vertically and each digit is printed in a 5 x 7 matrix. If the data is rearranged you can see how the up arrow character is formed from the data.



After this data is printed, the others will follow.



When the pointer in the print buffer reaches the End Of Line (EOL) character, it tells the PIK chip to stop printing and return the print head to the home position. The two [11] bits by themselves on the left side of the data will be ignored because the EOL would have been processed before the buffer pointer gets here.

At the moment, this does not make sense because the number 4 needs to be printed as well. The ACT does not stop processing instructions while the PIK is busy sending out print characters. The microcode will already be processing the data for the display formatted number 4, which in FIX 2 mode will be **4.00**. Remember also, that the printer takes time to print characters. The microcode will have already processed the data and have it ready in the C register well before the printer buffer pointer reaches the EOL character.

The next data will be sent out with a PIK1720 code. This mean the PIK will now interpret the received C data as 4 bits.

RegC = FFFFFFFFFE4A00 when CRC 1660 executes.

Arranging this into 4 bits we get:

```
F    F    F    F    F    F    F    F    F    E    4    A    0    0
1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 0100 1010 0000 0000
```

Inside the PIK, the 4 bit data will be appended with 11, so the internal data becomes:

```
1111**11** 1111**11** 1111**11** 1111**11** 1111**11** 1111**11** 1111**11** 1111**11** 1111**11** 1110**11** 0100**11** 1010**11** 0000**11** 0000**11**
3F     3F     3F     3F     3F     3F     3F     3F     3F     3B     13     2B     03     03
```

This data is added to the 16 character PIK print buffer and starts from the first EOL character, so the buffer will look like:

```
3F   3F   3F   3F   3F   3F   3B   13   2B   03   03   3B   38   00   1C   36
                                        ↑
```

Now we can associate each 4 bit field with a print character.

```
EOL  EOL  EOL  EOL  EOL  EOL  BLANK 4    .    0    0
```

So, after the PIK1660 Blank character is printed, then next data printed is 0 0 . 4.