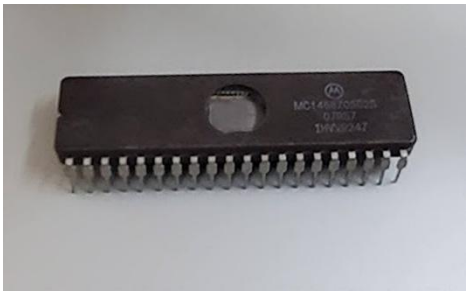
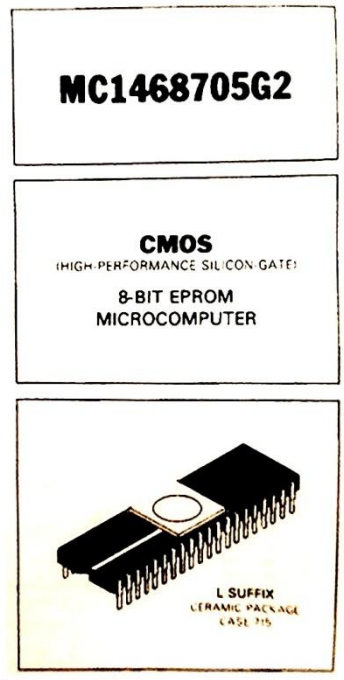
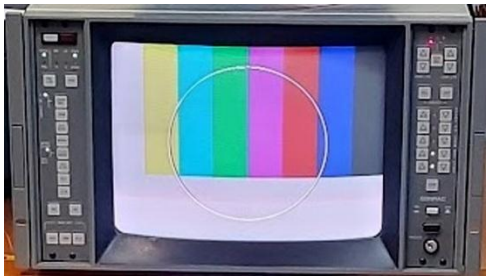
The MC1468705G2 MCU ROM Data Extraction Project.

Dr. H. Holden. Feb. 2024



The MC1468705G2 MCU.





Video Monitor with MC1468705G2 Systems Control

Background:

By the early 1980's some manufacturers such as Motorola decided that they should make a Microcomputer IC (or MCU) which contained its own internal RAM & ROM. It was a good marketing prediction, as this was the ultimate format for Microcontrollers of the future.

Prior to this, CPU IC's, examples being the 6502, 6800, 8080, or 8088 and others generally required peripheral support RAM and ROM to operate, notwithstanding their internal registers. Each function was isolated to a dedicated IC.

From the point of view of longevity of the equipment and for future repairs and restorations, this separate architecture is the "dream come true" Rom IC's could be removed from motherboards and dumped at will. This would preserve the code. Faulty RAM is easily replaced too. And, as most vintage computer restorers know, in general

at least, the CPU's themselves are usually fairly reliable, more so it appears than RAM and ROM IC's of a similar vintage.

The typical popular ROMs at the time were various types of UVeprom. These were wonderful to the extent that you could erase them with UV light and re-program them. They also have a very attractive clear quartz window, and you can see the IC die inside. Early date code types often had very large dies and are quite beautiful to examine.

However, not all forms of memory last hundreds of years and can fail in some decades, unless, carved into a stone tablet or Laser etched into Silica in the more modern version of the 10 Commandments.

The idea of putting UvEprom into a CPU or micro-computer IC seems appealing.

What if, one day, you have to get that data out? How do you do it, when provision was only made to put the data into the ROM by the manufacturers? No provision for exporting the byte file back in the other direction was made, back into an external memory of some sort.

And, say if it is 40 years on and you are worried about ROM rot, how can you make a backup spare part, or indeed make a spare part for somebody else when their original part has failed?

There is also no hope of getting the original ROM file from the manufacturer (I tried this) of the equipment in this case; the MCU and VDU it was used in is far "too old" to be supported. Probably the original data files went into the dumpster in the car park decades before, in the manner of the Apollo 11 computers.

Generally, MCU's like the MC1468705G2 are used in systems controls. Especially designs with touch buttons and key scanning and D/A conversions and with memory for settings. Thereby dispensing with the requirement for conventional front panel potentiometers & knobs.

In the case of the MCU in question, it was deployed in a 1980's vintage high quality color VDU made by Conrac. Conrac were America's premier CRT based VDU makers in California.

Manufacture of new CRT based VDU's has all but disappeared now. If this MCU IC failed due to ROM rot, the Conrac VDU would be rendered completely useless. Hence the requirement to want to back up this "Manufacturer Programmed MCU"

This article describes how the ROM file was extracted from inside a Motorola MC1468705G2 MCU

From this point, in this article, “MCUc” refers to the Conrac company programmed MC1468705G2 MCU in the VDU that I wanted to clone and thereby preserve its program.

“MCUt” refers to MC1468705G2 MCU’s which I had bought and programmed for the purposes of experimentation and helping to design the Data Extraction Machine.

THE MC1468705G2 MCU:

This device was described by Motorola as a high performance CMOS silicon gate technology 8 bit Eprom Microcomputer.

The MC1468705G2 MCU contains user programmable UV erasable Eprom.

It also contains an on board oscillator, CPU and RAM. One feature of it being, it contains an internal boot-loader program. The purpose of which is that the MCU can “program itself” without requiring an external programmer device, aside from some simple hardware.

(Perhaps not dissimilar to the notion on the TV series Dr. Who, where the Daleks constructed factories, where they could manufacture themselves).

The MC1468705G2 contains 112 Bytes of onboard RAM. 2096 Bytes of user programmable ROM at locations 080h to 8AFh, Including 11 extra bytes for programming a MOR (Mask Option Register and Vectors) residing in memory locations 1FF5h to 1FFFh.

This IC has 32 bidirectional I/O lines. The CPU was internally similar to the 6800.

The manufacturers set it up so that the user program bytes were placed in an external 8k ROM (such as a MCM68764 or MCM68766) in a ZIF socket on the programmer board.

This arrangement, with a fully UV erased MC1468705G2 MCU also on the programmer pcb, would, with the program boot loader activated, in program mode, transfer the bytes in the external ROM into the MCU’s internal ROM, a one way trip for the data file.

This programming event occurred when the reset switch on the programmer board was released.

The self programming was said (in the MCU’s data sheet) to take 200 seconds. Quoted at 100mS per byte. And the Verify program 8 seconds. I have yet to determine why this

was said. It is certainly not the case for the MCU's I have. Possibly these remarks relate to the early version MJ3 mask set variants only. I have not seen this variant.

With the MCU clocked by its usual 1MHz crystal, it takes only about 1 second to program the device and the Verify protocol another additional second, to check all 2107 bytes. At least, this is the case for the program that was in the particular MCU in the Conrac VDU.

Fortunately, the Verify program can be made to run separately, this saves time. It would have been possible to extract the bytes using both the programming and verify sequence together, disabling the MCU's actual Vpp programming voltage, however it would have doubled the processing time.

It appears that no program (firmware) was placed in the MCU to be able to export the ROM (Byte file) out of it, or at least nothing was documented by Motorola. There have been suggestions that this MCU contains an "undocumented protocol" to export the files. A Ghost in the machine perhaps.

In practice the byte file remains "trapped inside the MCU"

This sort of dilemma is going to be a problem encountered more and more as time passes. Many modern appliances and equipment run internal firmware, and the question is how to recover it?

The examples in this article are one method, for one particular MCU.

Clearly, if we want to repair and keep many vintage appliances (as they will be in the future) working, we will have to put our thinking caps on to figure out ways of extracting program data from many modern processor IC's. Especially, since in some cases, security locks also prevent it.

The purpose therefore is not to copy manufacturer's firmware for making rival equipment and sales, but instead the purpose is more noble and good. It is to keep vintage apparatus running. This helps prevent piles of E-Waste from discarded failed products when they could be repaired and give some more useful service life.

It is unlikely preserving this data from machinery that is decades old would greatly impact average sales of new goods, because, most consumers now who have adequate funding for new appliances have been groomed to believe that "new is good", so "out with the old and in with the new" as the old saying goes. We now generally live in a disposable society where most electronic goods are thrown away and replaced, rather than repaired. In a way this article also relates to the Right to Repair Movement, which is starting to gain some traction Worldwide.

Extracting the ROM Data from the MC1468705G2 MCU:

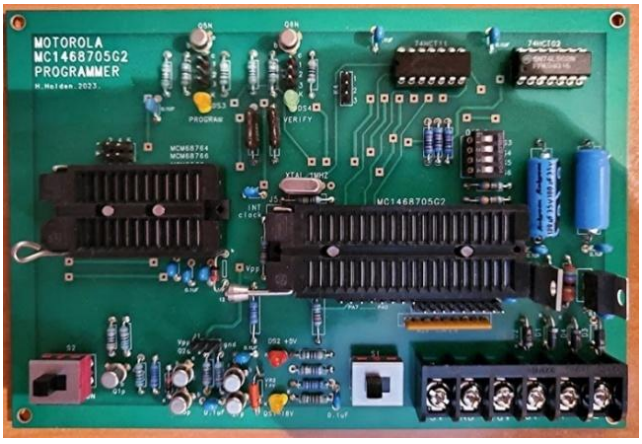
I realised one thing initially. If there was going to be any hope of extracting the internal ROM data, I would have to become familiar with programming this MCU. Motorola had designed a pcb in the 1980's, specifically for programming it.

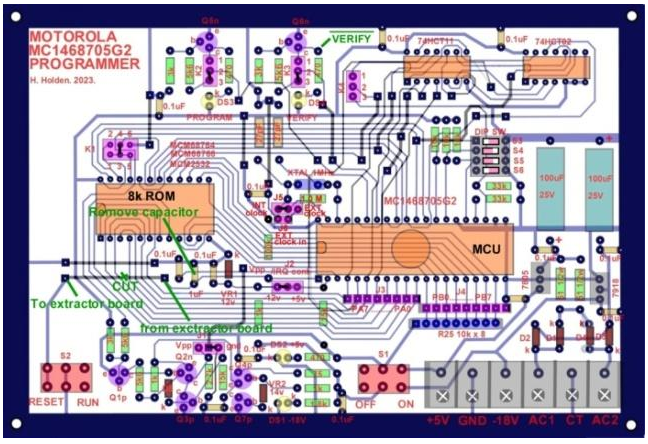
The reason being, if I was going to be 100% certain that I had extracted an accurate byte file from inside the IC I would have to know what it was beforehand.

In other words I would have had to program a MCU of the MC1468705G2 type myself, with a known byte file I created, so that I could check if the extracted file from a test MCU matched what I had initially programmed into it**.** "MCUt" from this point on refers to these test MCU's, while "MCUc" refers to the Conrac programmed MCU in the VDU, which I wanted to clone or replicate to make a spare part.

To this end I bought three new old stock MC1468705G2 MCU's and erased them ready for programming.

The first step in solving this problem was to build the Motorola programming board. The build of this board is covered in Motorola's AN907A Application Note. This App note includes the schematic and the PCB foil design. A photo of this board that I had made is shown below:





I altered the pcb design just a little with some extra jumpers, for example to run it from an external clock and to experiment with the /IRQ line and to ground the Vpp line to disable MCU's programming while conducting some experiments.

Also I added some headers to monitor the address lines, PA7 to PA0 and PD4 to PD0, which correspond to address lines A0 to A12. These were useful to monitor with a logic probe in the experimental stages of the project.

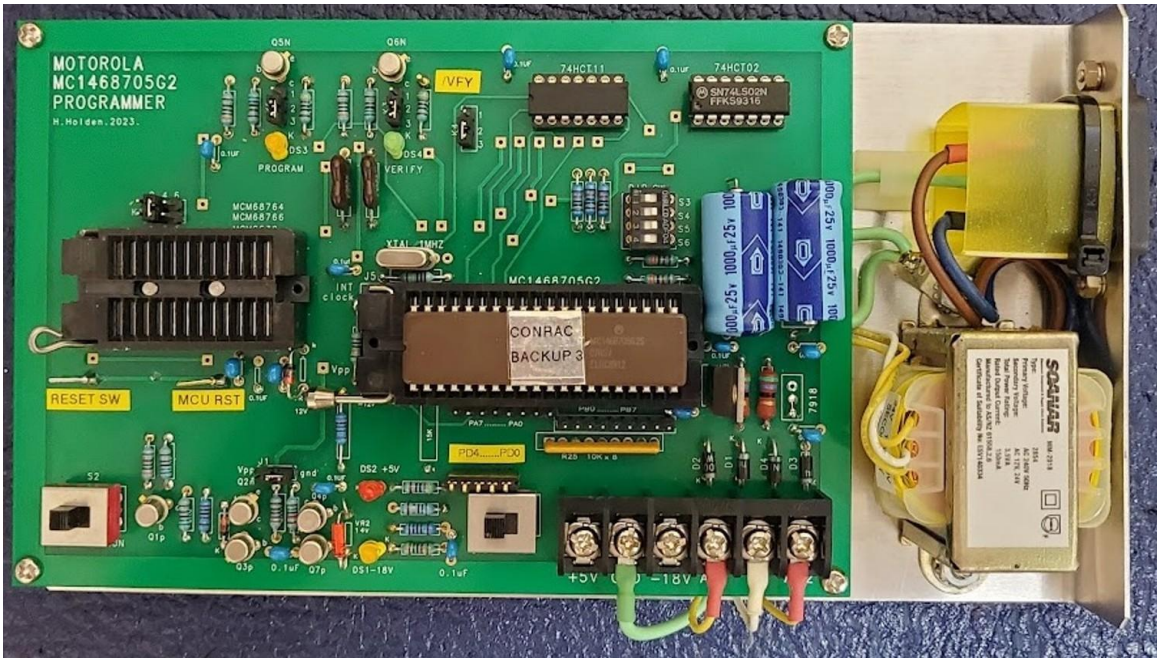
In the Program & Verify mode, the MCU is held in reset by S2 being closed. +5V and -18V power is then applied via switch S1 and then S2 is opened releasing the MCU from reset. Then the internal boot loader code is run, transferring the programming bytes from the 8k external ROM in the 24 pin socket, into the MCU. Then the Verify protocol is run. The DIP switch S3,S4 and S5 are all closed in Program & Verify mode. To run just the Verify mode, S3 is left open.

Power Supplies for the Programmer Board:

The board can be powered by a Laboratory +5V and -18 V power supply. It can also be powered by a Line operated 24V CT secondary transformer.

If a 24V CT line transformer is used, it is necessary to link out the -18V regulator IC. Also, because this board also has to power the “Data Extraction Machine” from the 24 pin ZIF socket, it required that the 100uF filter capacitors are increased to 1000uF.

The photo below shows the board mounted on an aluminium panel with an IEC Line power connector and a Line power transformer:



If the verification process completes, indicating that all of the bytes inside the MC1468705G2 match the file in the external ROM, then the green Verify LED is switched on by the MCU.

The link between the onboard reset switch and the MCU's reset pin was disconnected and capacitor C1 removed. These connections were then connected to the Data Extraction Machine Board. Along with a link to the Verify circuit, so as to disable the Data Extraction Machine's activity, once the ROM file in the MC1468705G2 has been fully sucessfully acquired by the Data Extraction board's NVram.

Operating Theory - Data Extraction Circuitry:

In use, in the Verify mode only, after release from reset (either by the reset switch or done electronically), the CPU marches through all of the addresses in a sequence starting at 080h (decimal 128), for 2096 bytes,and then it skips over unused erased state bytes to address 0FF5h to write data into the MOR (mask option register) and then 0FF6h to 0FFFh to write data into the Vector locations.This adds 11 bytes, the total being 2107 bytes.

If all the bytes loaded to the MCU's internal ROM match the external one, during the verify cycle, the verify LED is switched on by the MCU and the the address lines go to zero and stay there.

However, let us say a byte is defective and does not match between the MCU's internal byte file and the byte in the external ROM, the verify LED never lights.

The reason for this, as it turns out, is because the MCU stops (stalls) the verification process. The program stops executing. Mercifully, the MCU's address value stays on the exact address of the defective or mis-matching byte. It does not keep incrementing the address and it does not reset the address lines to zero in this condition.

The address of the mismatching byte stays on the address lines as a static value until a manual or electronic reset is initiated. This creates the opportunity for data extraction by an external device added in place of the external ROM, such as a ROM emulator. The idea being to keep changing the byte (with a trial byte) in the external rom, while repeatedly cycling the verify program, until the byte matches and then the verify program moves on to the next byte (at the next higher address value) and so on, until the entire byte file is verified.

The question then becomes, how long might this data extraction process take ?

Since it appears to take around 1 seconds from when the MCU is released from reset to verify the entire byte file, if a timer was set up to be just a little longer at say 1.2 seconds, and the verify LED signal did not occur (implying that the address counter is stuck on some address with a mismatched byte) the falling or rising edge of this simple timer, could initiate a sequence of events or a “Machine Cycle” (MC) referring to the activity of a data extraction machine.

With the external ROM replaced by non-volatile NVram (a Dallas DS1225 programmed with all bytes initially zero) the events of the MC would be:

1) Power up and hold MCU in reset and reset the trial byte counter to zero (though not definitely required).

2) Release MCU from reset and initiate verify sequence. If the timer timed out at 1.2 seconds and the file did not verify, this would imply a stalled MCU, stuck on some address. Then the process would be to uncouple the data lines of the DS1225 from the MCU with Tristate buffers and couple the DS1225’s data lines to the trial byte counter, again with tristate buffers.

3) Increment the 8 bit trial byte counter.

4) Latch the address value to a Hexadecimal display, to act as a progress counter.

5) Write to the DS1225 at its current address as specified by the MCU, which is the location of the mismatched byte.

6) At the end of the MC, initiate an electronic MCU reset & release, so that the MCU has another go at verifying the file with a higher byte value in the NVram location than the first attemt which failed. If the second attempt fails the MC (machine cycle) is repeated and so on.

7) Design Option 1:

As noted, If the verify light does deploy, then the 1.2 second timer is disabled and no more MC’s are generated. If the verify light does not deploy and the timer times out, then a machine cycle is generated and a reset and release deployed. In this article this approach is called a “fixed timer model” or **FTM**.

8) Design Option 2:

Arrange for a circuit to return to a reset & release directly from the mismatched byte, rather than waiting the whole 1.2 seconds. This can be achieved if an address Activity Detector Model (or method) called **ADM** in this article is used.

In both cases when, finally, the verify light deploys, the MC system is disabled and the file is verified.

I decided to go with option 2, to shorten the time to acquire the byte file.

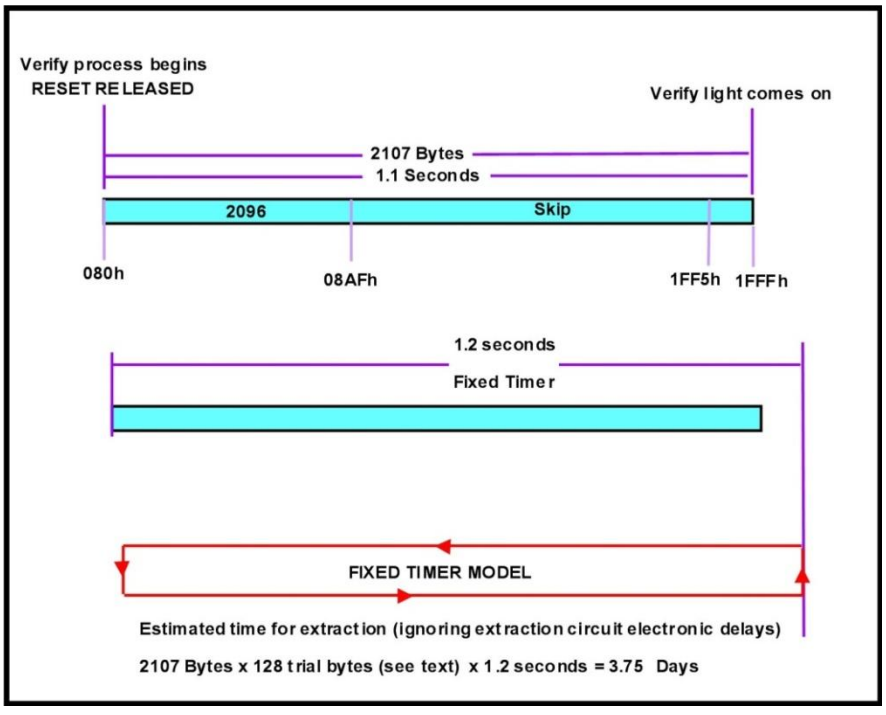
Considering the first byte examined at address128:

In the best case scenario the trial byte would match on the first or second attempt, if it were an 00h or 01h. In the worst case it could be an FEh or an FFh requiring a near full 255 counts or byte trials, in the case that the trial byte counter started from 00h.

Some bytes of course will occur more frequencly than others, some being OP codes, others will be immediate data. This is a very rough approximation. The trial byte counter doesn't have to start at 00h (in fact in this design it is only zero'd at power up) as long as it gets to cycle though all possible 256 byte values, if it has to, to make the byte match. Therefore it is set up as an overflow counter.

It appears then, that ***on the average*** the number of bytes that will be trialled to gain a byte match at any address will be very roughly (best case + worst case) divided by 2, or 1 trial best case versus 255 more trials worst case or (1+255)/2 or aproximately, an average of 128 trial byte attempts per a successfully matched byte.

The first design option 1 is depicted in the diagram below. A Fixed Timer Model (FTM). Essentially if the time to normally switch on the verify light is exceeded, the MCU is deemed to be stuck on a mismatched byte. Clearly, using the above rough calculations, this method would be a little long to be very practical, but it would still work. Taking very roughly 3.75 days to complete. With a byte recovery rate of 2107 bytes per 3.75 days (90 hours) at roughly about 23 to 24 bytes per hour acquired.



However, with a slightly more elaborate circuit than a fixed timer, rather than having to wait until just after the verify light should have deployed, to determine if the MCU's address counter was stuck, this system checks for changing activity on the lower address lines A0 and A1.

If the activity stops for a period (which is a longer than it takes to skip over the address, when the byte at the address verifies normally) this generates the MC (the machine cycle) to increment the trial byte counter, write to the NVram and pulse reset the MCU for another trial.

This system, an Address Detection Model **ADM** is the one I decided to go with. It is depicted below:

With the ADM model, if address activity stops, the MCU is deemed to be stalled on a byte mismatch. There is a rapid reset, the trial byte in the NVram incremented and the process repeats. With this model, for at least the first 10 hours, the early part of the byte data acquisition is dominated by delays in the detector circuit generating the machine cycles (MC's).

Conrac did not use all the user programmable address space in the MCU. The usable range id 080h to 08AFh, they used 100h to 7F9h, and left the rest of that user area as 00h. And there were some zones in the area they used that were not programmed (all 00h)

Of course, toward the end of the file, from at least about 1200 bytes along to the 2000+ byte area, the time between MC's would start to approach the 1 second period, similar to the FTM option, because of the time that the MCU verify protocol has to count up to the high range address values in that region. Therefore, when verifying bytes at the end of the file, in theory at least, it could be nearly as slow as the 24 bytes per hour of the

FTM. I found the slowest rate near the end of the byte file acquisition to be around 28 bytes hour with the ADM.

I found though, that using the ADM with the electronic delay constants I had built into it, that the system appeared to be matching around 200 bytes per hour in the early phase of the file.

The early figure of 200 bytes/hr is dominated by the timing delays in the two Mosfet circuits and the values of the coupling capacitors in the address activity detector. With a smaller contribution from the MCU verify protocol. The figure later on, after about half the file is acquired, is dominated by the delay in the verify protocol itself.

It also turned out, with the particular MCUc's byte file on testing and starting with a blanked Dallas Nvram, it took 26.2 hours to completely process.

While I could have shortened the time constants on the data extraction board, to speed it up, in the initial phases of the byte recovery, it became clear that ultimately, it was the speed that the Verify firmware runs, which is the main determinant of the total delay. I was much more interested in reliable byte recovery, than the total time taken for the file extraction.

The rate of bytes resolved (or acquired) per hour as time passes, for the ADM becomes an exponentially decreasing process.

This is because each time a successful byte match is made it takes longer for the verify cycle to get to a higher address to reach the next mismatched byte along in the chain of bytes. It is somewhat analogous to a bouncing ball where its bounce height decreases by a fixed percentage with each bounce and it results in a logarithmic bounce height decay.

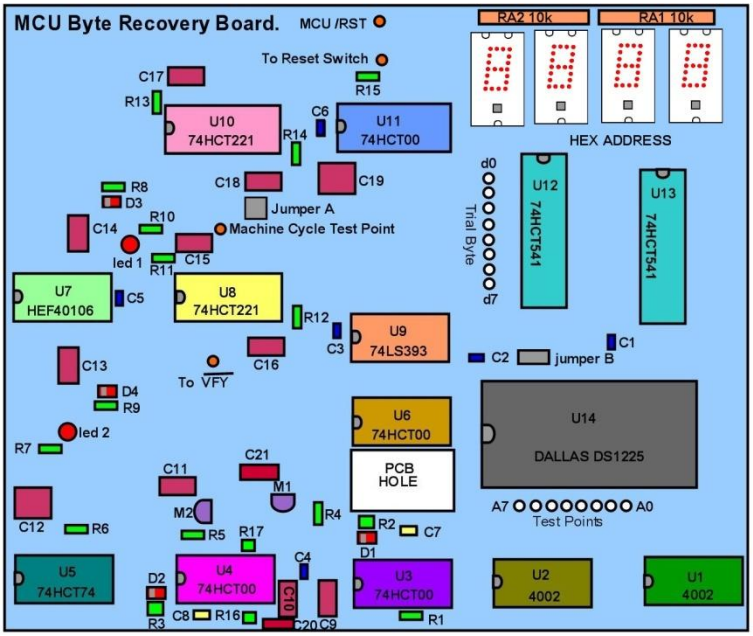
The Data Extraction Board:

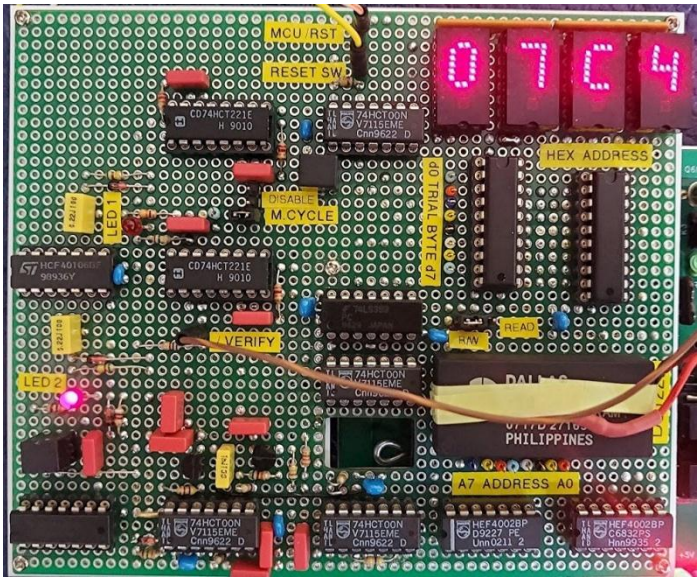
The data extraction board was arranged to plug onto the 24 pin ZIF socket on the Motorola Programming Board.

(Of note: In a project like this, a lash up on an experimental board could be a disaster, because even one intermittent connection could foul up the long time frame experiment. Therefore it was built on plated through hole pcb material with each connection carefully soldered).

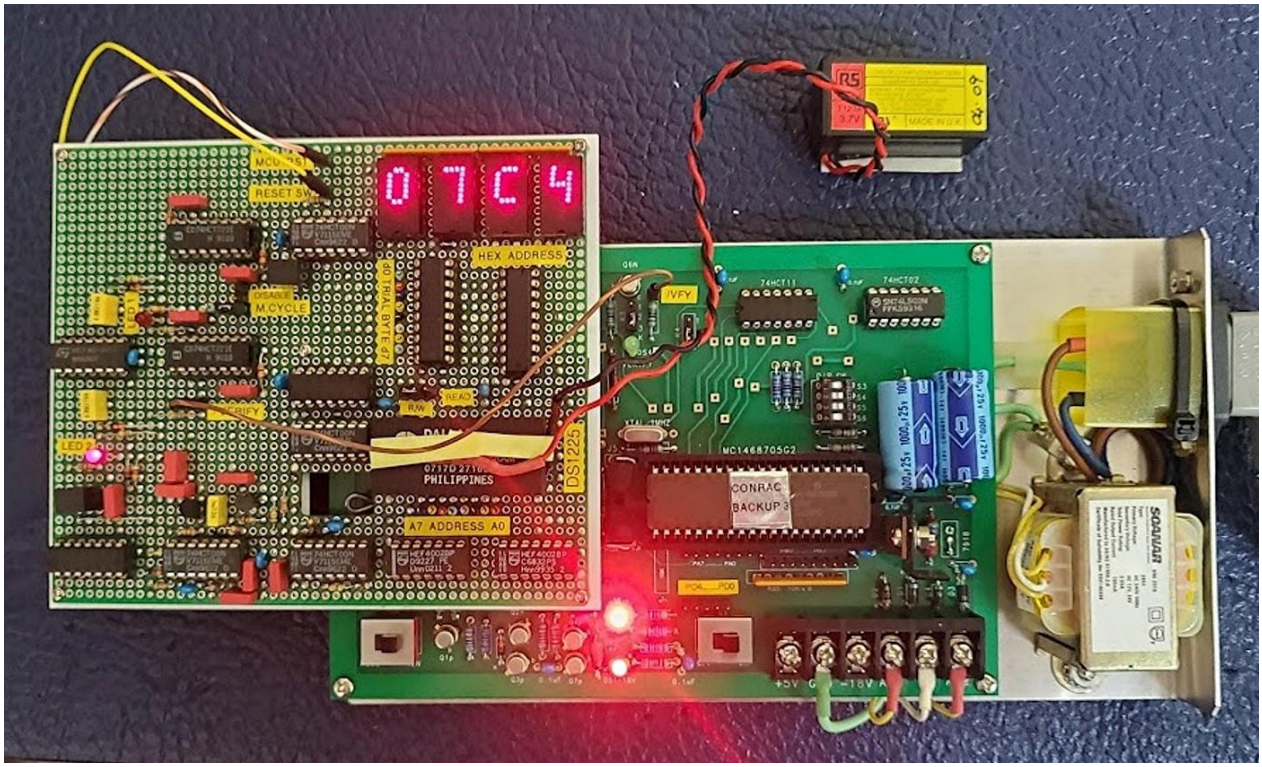
The Topographical layout is shown below. The wiring side of the board was then covered with 1mm thick Styrene sheet to prevent any accidental sort circuits. Long Gold

plated pins were used to engage the 24 pin ZIF socket on the Programmer Board. The Data Extraction Board acquires its 5V power from the ZIF socket too.





The photo below shows the data extraction board plugged onto the ZIF socket on the Programmer Board. Because the access to the 24 pin ZIF socket's Release Arm was required, the Data Extraction Board required a rectangular hole.



Most of the DS1225 NVrams that I have in my stock have discharged internal Lithium batteries now.

You will notice that the DS1225 I have plugged into the board has an external lithium “support” battery applied. This is connected by milling the plastic casing down to the + battery terminal in the module, which, in this variant, is on the module’s top. Many other DS1225’s have the battery in the bottom and it is more awkward to gain a connection to the battery’s + terminal. The negative terminal is pin 14 of the IC.

Also, with the additional current consumption on the 5V rail, it is worthwhile adding a heat flag to the the 7805 voltage regulator on the programmer board (this can be seen in the photo above). Also, as mentioned, the two 100uF supply filter capacitors are increased to 1000uF 25v units.

There are three plugs on wire links between the two boards. One is from the Reset switch on the Programmer Board (it was isolated from the MCU reset line by cutting the link track) the other is a feed to the MCU’s reset pin.

One other connection is to the collector of transistor Q6 on the Programmer Board. This line goes low when the green verify LED activates and it inhibits the Machine Cycle of

the Monostable U8 pin 3 on the Data Extraction Board. Capacitor C1 is removed from the Programmer Board.

The schematic below shows the Machine Cycle Generator. Its job is to identify when the verify protocol is being executed by the MCU, and when that process has stalled, due to a mismatched byte.

Ultimately, when all bytes in the MCU's ROM match those in the Nvram, then the process is complete and the verify light comes on.

The Dallas Nvram at the end of the process holds the "byte record" of what is in the MCU's internal ROM. This is exactly the same byte file that was used to program the MCU in the first instance. Essentially this tool reverses the function of the Motorola Programmer Board and ultimately puts the MCU's byte file back into the external Nvram.

After the process is complete, the DS1225 is simply transferred to a GQ-4x Rom reader, or similar, and the file recovered and saved.

However, the DS1225, as it is, now containing the correct data, can be used to program a fresh blanked MCU.

To do this, there are jumpers on the data extraction board. Simply the machine cycle circuit is disabled by jumpering the MC signal that feeds the memory controller to ground with Jumper A. The /WE pin on the DS1225 is jumpered to +5V with Jumper B.

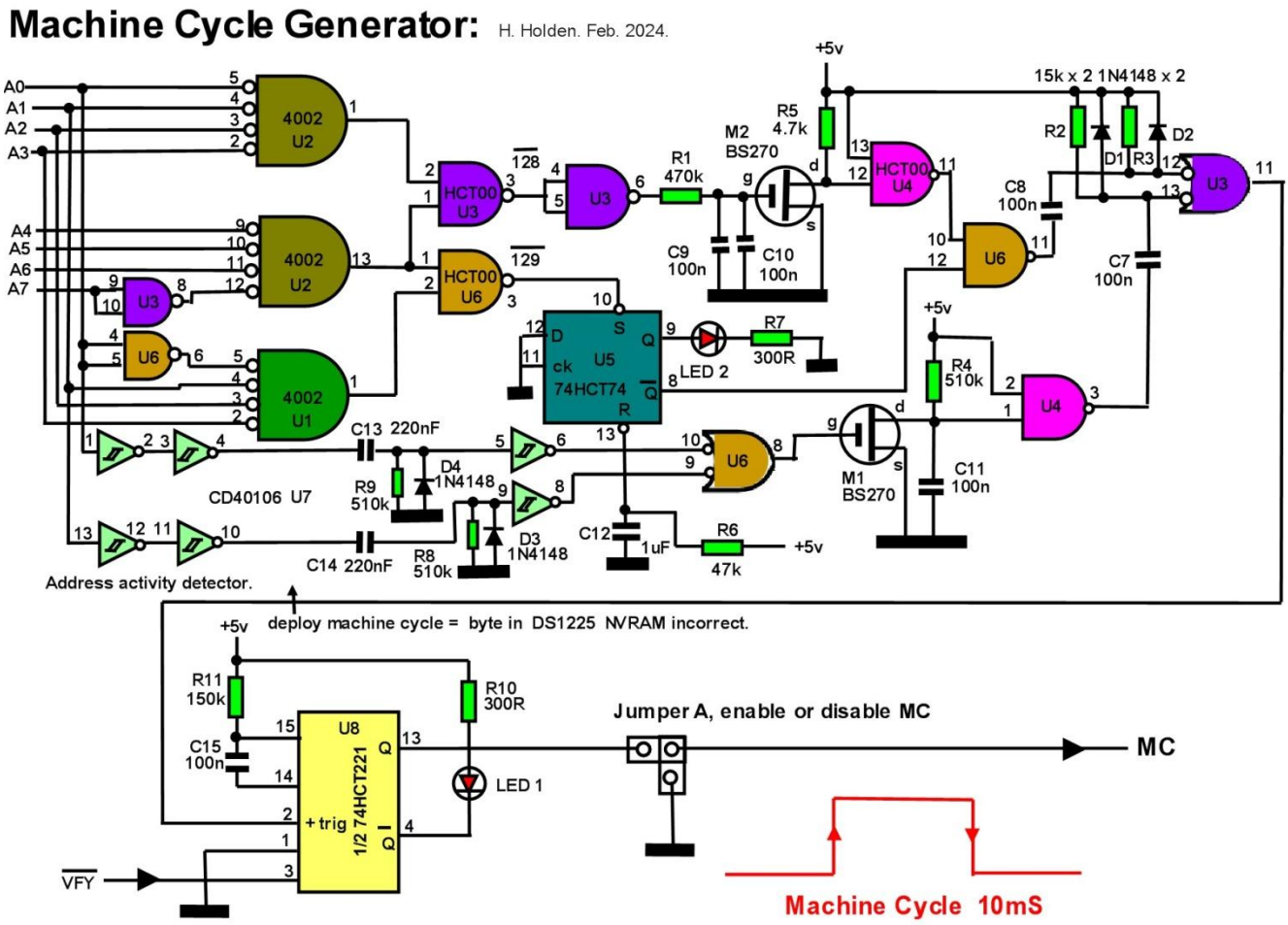
And the Vpp pin in the MCU on the programmer board is jumpered away from ground and to its normal connection (with the added jumper J1 there). And switch sw-3 on the programmer board is turned on, to enable the Verify and Program mode.

The DS1225 on the data extraction board, then acts as a "stand in" for the usual ROM with the source data, that one would have had in the 24 pin ZIF socket for MCU programming purposes.

Delay timers, based on BS270 Mosfets were used to help make sure the conditions were stable and give perfectly reliable detection of the absence of address activity, but not such a fast response time that correct sequential bytes are prevented from being stepped over as the verify process proceeds. For example, if the file does verify no MC pulses are generated. The system starts with the DS1225 blanked (all bytes = 00).

One of the issues affecting the detection of dynamic address changes is that, initially at least, the MCU, deploying the verify cycle, starts at the address decimal 128 or 080h. In this case, both of the address lines A0 and A1 remain low and there is no loss of dynamically changing signals to detect. This required that address 128 be decoded to

create a machine cycle if it remained stuck initially on 128, which is likely the case, if the first byte in the DS1225, which is initially zero, doesn't match the one in the MCU's ROM at the decimal address of 128.

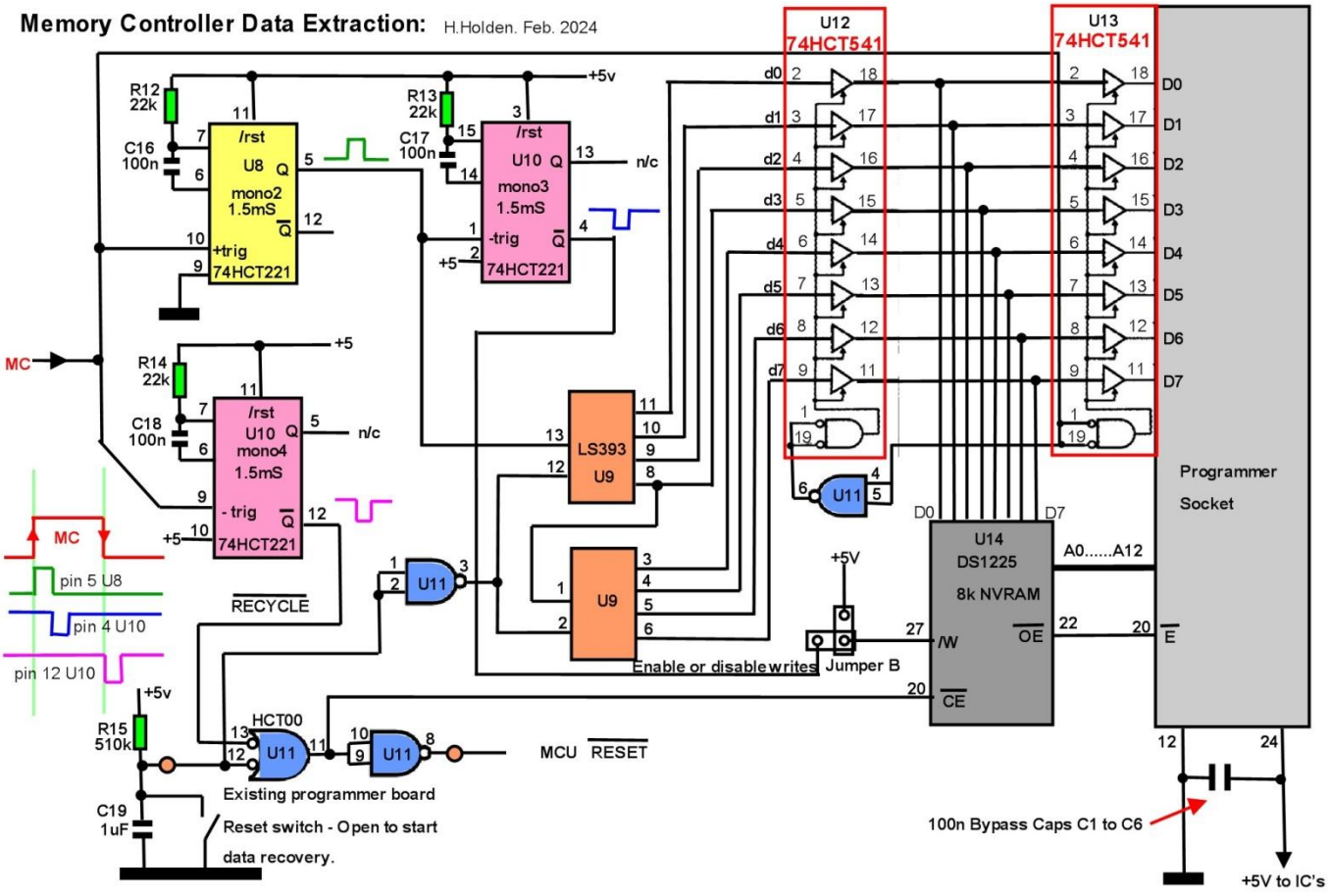


After that though, the value 128 must be completely ignored. This could have been done two ways, one by using full decoding of the value of 128 (080h) by decoding the upper address lines A8 through A12. I decided to do it a different way, by decoding address 129 and using it to set a flip flop.

The reason being, then I could inhibit the 128 signal from that point of addresses upwards, on U6 pin 12 and I could use the flip flop output to light an LED from its Q output terminal. This LED illuminating indicates that everything is working and that the first byte was made to successfully match and remains safe & sound at address 128 in

the Dallas Nvram. Later though I added the Hex displays which make it very easy to monitor the byte acquisition process visually.

The MC signal drives the memory controller circuit shown below. Once a byte mismatch is detected, because the address activity has stalled, the trial byte counter is incremented and this byte value is written to the address (that the verify program stalled on) into the Nvram. Then the system is auto pulsed reset by monostable 4 U10 pin 12. This allows the MCU to have another go at matching the new byte at that address in the Nvram.



Monostables U8 & U10 generate the required pulses to increment the trial byte counter (based on a 74LS393) and write to the Dallas NVRAM and provide the reset pulse after the MC is complete.

One interesting consequence of the design is that when the maker of the ROM file (Conrac) did it, they left a number of zero bytes in a row. They had zeros from address 80h to FFh and they started their program file at 0100h.

With the bytes in the Dallas Nvram being all initially 00, the initial block of 00's verifies very quickly as do a number in the trailing end of the user program area.

The MC signal itself is used to Tristate the DS1225's data outputs, to either the MCU normally or to the output of the trial byte counter while the MC pulse is still in progress.

The MC pulse could be shortened to 5mS, but calculations suggest it would only shave less than 1 hour off the processing time. Most of the delays are in the time it takes the MCU to run the verify program after resets, especially as time passes and over half of the file is resolved. Even if the MC pulse was shortened from 10mS to 1ms wide and the other monostables cut back from 1.5mS to 300nS each, it would only reduce the total process time by less than 1 hour. I used the relatively long pulses from the monostables, in the 1.5 millisecond range, so as to be 100% sure everything was stable for the NVram writes.

Hex Address Display:

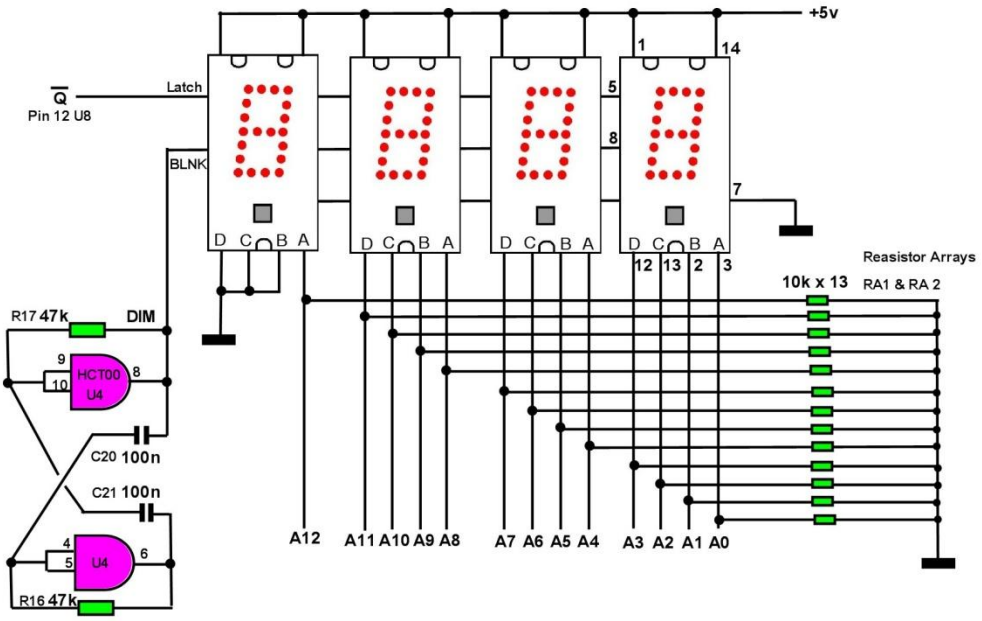
Since it takes a moderate amount of time to acquire the bytes from the CPU, a progress indicator of some kind is valuable. I decided to use Hexadecimal displays to show the Hex address of the byte undergoing the matching process.

This can be made stable because the display modules have internal data latching. I used the /Q output of monostable U8 pin 12 to latch the current address undergoing machine cycle matching into the Hex displays.

The display modules I used are the functional equivalent of the famous Texas TIL311, but they are not TTL, but low current Cmos versions. They are the Innocor INL0397.

I noticed with the address lines, A0 through A12, when the verify process has not started, or is complete and all of the lines are supposedly zero, these lines are in fact tri-stated open circuit by the MCU. Because of this I added 10k Pull Down resistors on these address lines, to make sure they were not susceptible to noise pickup in that state, and to be sure, that after power up, before the Reset was released, that the displays showed 0000h. To reduce the display brightness and current consumption, the display modules are blanked 50% of the time from a square wave oscillator, provided by two spare gates of U4.

The circuit of the Address Display part of the schematic is shown below:



Unexpected findings:

There turned out to be some interesting vagaries of the MC1468705G2 MCUc which I was wanting to clone, versus the MCUt units I had trialled for experimental purposes the MCUt's:

1) Not known to me, the ROM bytes in MCUc from decimal 128 to 255 (0FFh) were all programmed by Conrac as 00h. They were probably planning to save some space there for applications later.

Therefore, with the initial Dallas RAM programmed for all zeros, these all immediately verified right up to decimal byte number 256 (100h) where it encounters a mismatch. Because the dynamic address detector, monitoring activity on address lines A0 and A1, was previously active, before it stopped on address 256 (unlike the case where the MCU protocol started initially on 128) then it turned out that for this particular MCUc's program, I would not actually have needed the 128 and 129 fixed address detectors at all. But of course, I would have to have had a Crystal Ball, to know what the byte file in MCUc looked like in advance. In the design every possibility of byte value had to be allowed for. Therefore the 128 & 129 dedicated address detectors are still required for the general case.

2) This next issue was very perplexing initially, before I figured out what was happening.

The MCUt devices (having similar markings to the MCUc and similar date codes and none of them being the alternative early MJ3 mask set versions) would all initiate the verify protocol when released from reset and in the verify mode, **without** the -18V applied to the programmer board. This applies a Zener regulated -14V to the MCU's /IRQ PIN 2.

I preferred to keep that -18V supply disconnected. Also I had permanently grounded the Vpp pin 3 of the MCU with a jumper J1, and only had the DIP switch SW3 set for “Verify Only” mode, sw3 being in an open state. Being “triple insurance” against accidentally damaging the file (programming) the original Conrac MCUc with an unintentional programming event.

When I put the precious Conrac MCUc into the programmer board and released it from reset (as I had done for all the other MCUt’s I had been using) it sat there doing nothing and the Verify program did not execute, unlike the other MCUt’s….Gulp !

This was a somewhat horrifying moment after all the work I had put in to design & build make the data extraction system. After some experimentation I found, MCUc requires that it has -14V applied to the /IRQ pin 2. If not the Verify process will not start.

The other MCUt’s all started the process with that pin at zero volts, this is until I programmed them with the recovered Conrac byte file and it became clear what was going on.

The MCUt’s when programmed with the Conrac file then also required the -14V supply on their /IRQ pin, to behave as the original Conrac MCUc does. The reason was the MCUt’s that I was using, I had programmed them with random byte data for my experiments, including the MOR Register and this affects how the MCU responds when released from reset.

Project Outcome:

At this point I have been able to program three blank MCU’s with the orginal Conrac file and they all work perfectly on testing in the Conrac VDU.

This Data Extraction Machine is possibly the only one currently in existence for the MC1468705G2 MCU.

The method outlined here may work for other similar MCU’s. There is enough detail presented in this article for anybody to build their own data extraction machine.

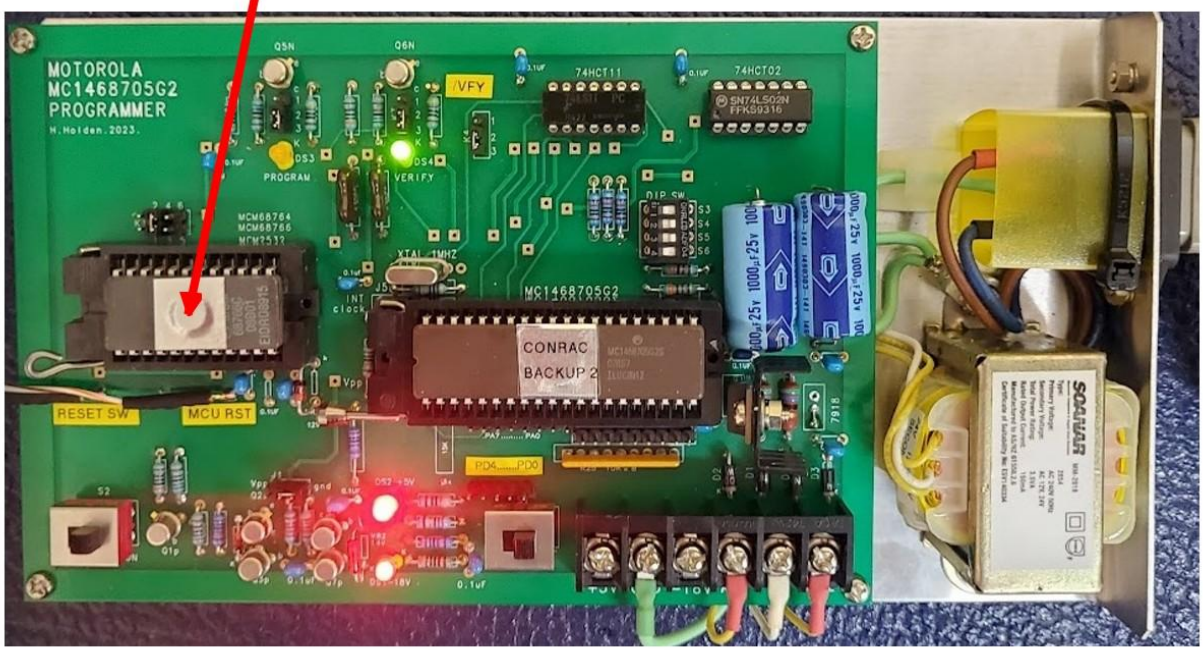
The design is proven to work and is composed of very easy to get vintage logic IC’s and requires no computer or firmware assistance to operate it. It is required that the constructor also builds the Motorola Programming pcb, though the entire project could be built onto one pcb.

To finally come full circle, I programmed the data file retrieved from the Conrac programmed MCU into a pair of vintage MC68766 UVeproms (recommended by Motorola) as the external ROM device on their programmer board. To do this I had to

deploy my vintage BP Micro-Systems 1400 Programmer as this UVeprom appears not commonly supported by new generation programmers.

This programmed ROM now represents what Conrac would have had in their factory in the 1980's era. It verifies with the original MCU from the VDU and also with the three replica MCU's that I made, as spare parts:

MC68766 - Programmed with retrieved file & Verifies



****