CoCo~123

Glenside Color Computer Club, Inc. Carpentersville, Illinois

Volume 35, Number 2 Summer 2015

CoCo ~ 123 Memory Map

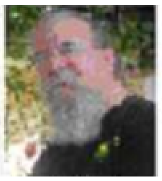
Upcoming Events:
The 25st Annual "Last" Chicago CoCoFEST! **April 23-24, 2016**
Regular meetings every 2nd Thursday @ Schaumburg Library

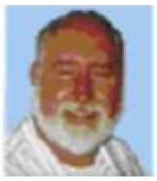
G.C.C.C. OFFICERS

Here is the list of 2015 club officers and how to contact them. If you have questions about the association call one of the officers for the answers.

POSITION	**NAME**	**PHONE**	**PRIMARY FUNCTION**
President	Tony Podraza	847-428-3576	The buck stops here...
Vice-President	Chris Hawks	815-943-4690	Meeting planning, etc.
Treasurer	George Schneeweiss	815-832-4441	Dues and Purchasing
Secretary	Bob Swoger	224-236-5194	Records and Reporting
Director	Tony Podraza	847-428-3576	CoCoFEST! Organizer
Librarian	Brian Goers	708-805-1888	Special Events
Secretary	Bob Swoger	224-236-5194	Records and Reporting
Editor	John Mark Mobley	847-409-8604	Newsletter Production

Tony Podraza


Chris Hawks


Robert Swoger

George Schneeweiss

Brian Goers

John Mark Mobley

CoCo~123 - A Glenside Publication Since 1985

CoCo~123 INFORMATION

CoCo~123 is the newsletter of the Glenside Color Computer Club. Your annual contribution of $15.00 helps to keep our club going. Send your check to Glenside Treasurer:

George L Schneeweiss
13450 N 2700 E Road
Forrest IL 61741-9629

Our treasury provides newsletters and good times with fellow CoCo users at our Annual "Last" Chicago CoCoFEST! and Annual Glenside Picnic.

CoCo~123 CONTRIBUTIONS

If you have any suggestions for the newsletter or would like to submit an article, please contact the CoCo~123 Newsletter editor:

John Mark Mobley
4104 Wren Lane
Rolling Meadows IL 60008
johnmarkmelanie@gmail.com

CONTRIBUTORS TO THIS ISSUE

Richard Bair
Kip Koon
John Mark Mobley
Tony Podraza
Richard Goedeken
George Schneeweiss
Robert Swoger

G. C. C. C. MEETINGS

The Glenside Color Computer Club meets the second Thursday of each month at the Schaumburg Township District Library at 7:00 pm. If you need a map, see our Glenside Homepage at:

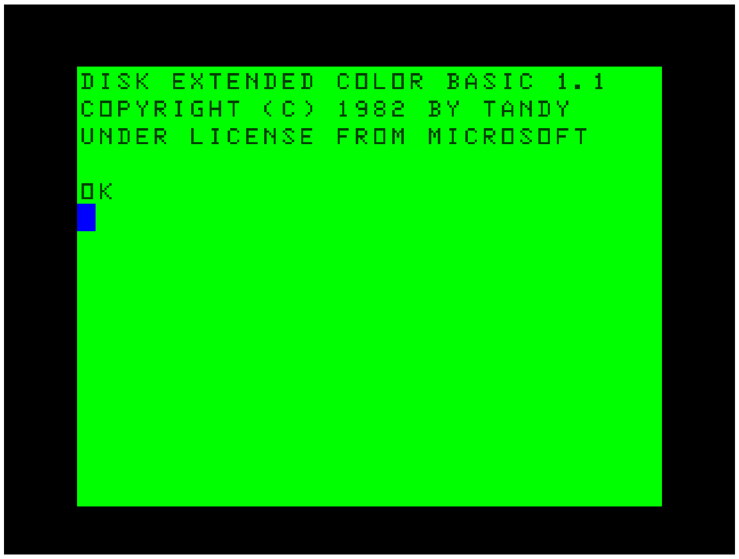
http://glensideccc.com/splmap.html

A social get-together always follows the meeting at a nearby restaurant, lovingly called, "The Meeting After".

FROM THE PRESIDENT'S PLATEN

Falling down is easier than getting up. That is a fact due to gravity.

Growing older is a fact of life; growing up is optional. For many of us in the CoCo world, growing up, with respect of computing, may have been forced upon us by the tools that we use at work and the tools that they require to be used. Yet, we still gather together to share ideas and show off our newest CoCo-related hardware or software. There are all types of hardware that are being used as peripherals for the CoCo. Indeed, the 6809 code has been ported to other platforms, and the hardware that it uses is widely varied. But, at heart, CoCo-ists are still young. We don't necessarily revel in the CoCo...but there seems to be years disappearing from our composure and visages whenever we see , "



```
DISK EXTENDED COLOR BASIC 1.1
COPYRIGHT (C) 1982 BY TANDY
UNDER LICENSE FROM MICROSOFT

OK
```

Next year, Glenside will, once again, host a "Last" Annual Chicago CoCoFEST! Twenty-five years running. It is not Glenside that has kept the FEST! running, it is you, those who drive countless hours, stay up until 5 in the morning, soldering the last few boards to have them to show off and, perhaps sell enough for enough gas money to get back home; but most of all, endure all this to enjoy a few moments in the company of others who are also captivated by the microprocessor that has held us in its grips for many years, some of us from the time that we were not grown-ups....but then growing up is optional.

Till next time, I bid you Peace.

Tony Podraza, President
Glenside Color Computer Club

TREA$URY NOTE$

We have $6020.34 in checking. We have 88 paid members in 2014, down from 136 in 2011, and a total of 431 members to whom we send newsletters.

George Schneeweiss, Trea$urer
Glenside Color Computer Club

THE SECRETARY'S NOTEBOOK

Our gas meter in our garage blew up Nov 12, 2014. We finally got back in the house in June after 7 grueling months in an apartment. So please forgive me for not contributing to the NL as I have been fighting the insurance company to get re-reimbursed for all we lost. I have spent weeks every day for 8 to 20 hours a day in front of the computer to submit receipts to the insurance company. I feel I'll be at this until the end of November. Thinking of you all. I pray I'll see you at the next fest. I must also thank the Glenside members for providing a wonderful Annual Picnic at our home this year. I'm sure you will get a nice report on the picnic in the Autumn NL.

Bob Swoger, Secretary - rswoger@aol.com
Glenside Color Computer Club

THE EDITOR'S CLIPBOARD

I need help getting the newsletter out. I am starting a new job and may not be able to continue the newsletter. Let me know if you can help. The Newsletter Editor must be appointed by the President.

John Mark Mobley, Editor
Glenside Color Computer Club

CoCo3/HDB-DOS/NitrOS9 Adventures, Part 3 of 3

by Rich Bair

I ended my last column with a promise of an explanation of my NitrOS9 boot failure problems. The short answer is, there appears to be a maximum size for the OS9Boot file, and it's not directly related to the amount of free system RAM.

A lot of the clues I found were learned from studying the mdir -e displays of the boots that did succeed. The mdir list always starts off with the three modules that are read from track 34 of the (real or virtual) floppy. They are written (after some fudging) into the highest of the 64 8K RAM blocks on a 512K CoCo, number 3F (in hex), which is then mapped into the top 8K of the system address space just like in DECB. But remember, the 18 sectors of track 34 add up to only 4 1/2K of data. So as not to waste system memory space, NitrOS tucks the three modules right up under the GIME/IO page, leaving about 3K still unused at the bottom of the 3F memory block.

Mdir doesn't mention this, but at this time the lowest 8K-memory block, number 0, is mapped into the bottom of the system space to contain some of the various system data structures.

Once the track 34 modules are in place, the boot module looks at the size of the OS9Boot file to decide how many more blocks of RAM (maximum 4) to map in for loading it. These will start with block 1, since 0 is already taken for data, and will be mapped in as a unit right under block 3F.

So the mdir -e list continues with a list of the modules in your bootfile, in the same order that you arranged them. At the left of each item is the block number 1, but if you pay attention to the addresses and sizes of the individual modules, you will realize that by about the fourth module you've moved out of block 1 and into block 2. Mdir gives no hint of this, even as you move into block 3, and most likely 4, and, remarkably, even into the unused portion of block 3F! They're all called block 1 by mdir.

The mdir -e display is also a bit confusing because the system blocks are listed from high system memory locations to low, but the individual modules are ordered from low to high. So the last module in the "block 1" list actually is adjacent (more or less) to the first item in the "block 3F" list above. Depending on their sizes, the last several items in the "block 1" list may actually be in block 3F!.*

At this point you may be calculating "OK, 4 blocks in 'block 1' equals 32K, plus the 3K unused in block 3F gives me a bootfile max of 35K (=$8C00)". I wish. But another look at mdir shows that the bootfile doesn't start at the very bottom of block 1. This is partly because NitrOS9 purposely tucks the whole bootfile as high up in memory as it can without bumping into the modules already in block 3F. But in all except possibly one** of my boots that have succeeded, I've never been able to get the bootfile modules to start lower than $0A00 in block 1, which takes away about 2 1/2K. In my experiments the size maximum appears to be $8306, or about 32 3/4K. This limit is not directly related to limited system RAM, because the successful boots that are only a page or two below the limit show 30 or so free pages of system RAM (7K+).

Because of this size limit, none of my successful boots have contained all of the modules that I originally planned to put in a single bootlist. So the remainder of this discussion will deal with some of the issues involved in making multiple NitrOS9 boots a reality.

Setting up multiple DOS boots in HDB-DOS might at first seem like a non-problem, because HDB-DOS does accept a drive number parameter following the DOS command. But remember that a bootfile on a virtual floppy must be linked to the hard drive first, and the DOS command doesn't do that. So if yesterday you linked the bootfile on virtual drive 0 to the hard drive and today you try a DOS 1 command, you're still going to get your drive 0 OS9Boot file (after loading track 34 from drive 1).

The next thing you might try is an autoexec program located on each drive that links that drive's OS9Boot. But the problem with that attempt is that in HDB-DOS an autoexec file only runs on a DOS command if there is NOT an OS boot on track 34 of that particular virtual floppy. It would have been nice if the autoexec file would kick in first, but it doesn't.

So here's my solution: Remember from my earlier column that I planned to have an autoexec on drive 0 that presented a menu screen on bootup. (This means I do not put a NitrOS9 boot on drive 0, or the autoexec won't run.) Then my menu presents choices such as "Press 2 to boot NitrOS9 from virtual drive 2", and if that option is chosen the menu program runs a short program on drive 2 which links*** its bootfile and then issues a DOS 2 command. Now the number of different boots I can have is limited only by the amount of space on the menu screen.

At this point another problem crops up. Different boots are likely to work best with different startup files, but nitrOS9 expects that file to be in the hard drive root directory, so how can you have more than one?

The solution to this one is a bit more complicated. The module that calls startup is sysgo, which is often included in the OS9Boot file although it can be put in the /dd root directory to save system memory. So at the cost of a slightly larger bootfile, I created modified versions of sysgo to include in the boot. Their files are named sysgo2, sysgo3, etc. in my directories, but the module name is still sysgo. Instead of looking for "startup", these modules look for "startup2", "startup3", etc., so that now there's no limit on how many startup files I can have available in the hard drive root directory. Sysgo is also the module that looks for an autoex file. I haven't done it as of this writing, but the same kind of modification could have each different boot look for its very own autoex file.

Incidentally, the sysgo module is also the one that puts that scary message on the screen that says "** DEVELOPMENT BUILD ** ** NOT FOR DISTRIBUTION! **". While I was in a modifying mood, I changed that text in sysgo to something more friendly.

So my CoCo is now back to functioning in a way that feels good to me. I'm sure challenges still await, but for now I'm done with major modifications. If I have made statements that you know to be wrong, or if you have better solutions to some of these problems, please let me know. I'll be happy to write up corrections or additions.

* After the boot process is finished and several processes have requested and then released memory, it gets even more complicated. A module larger than 8K may be loaded into blocks that are not even consecutive numbers. Mdir tells you only the number of the starting block. But if the module is an active process, pmap will tell you the rest.

** At one point in my investigations when I wasn't documenting my steps very well, I printed out an mdir -e that plainly shows a bootfile that loaded starting at $0100 in block 1. That bootfile has all the same modules as one of my successful boots that loads at $0A00. I don't know what I was doing differently that time, and I've been unable to duplicate that result. If any reader can give me any clues as to what could cause that difference I would be very interested, as it could potentially allow a bootfile 2 1/4K larger than my apparent maximum.

*** The LINK.BAS program distributed with HDB-DOS has two functions, and is designed to be used only once on each floppy disk. Besides linking the bootfile (it assumes the floppy is already os9gen-ed) it creates an RS-DOS partition and directory from track 16 up (except for track 34). Strangely, though, it doesn't tell the OS9 partition that half its space has been taken away. (I added lines to my full version to correct this oversight.) But for the purposes described above, only a small portion of the program is needed. I leave it to the reader to study the BASIC program and find the lines that accomplish the linking.

Rich can be contacted at mgdoc1@sbcglobal.net

Real Programmers Code of Conduct
Submitted by: Kip Koon
Author Unknown

- Real programmers don't write specs -- Users should consider themselves lucky to get any programs at all and take what they get.
- Real programmers don't comment their code. If it was hard to write, it should be hard to read.
- Real programmers don't write application programs, they program right down on the bare metal. Application programming is for feebs who can't do systems programming.
- Real programmers don't eat quiche. They eat Twinkies, and Szechuan food.
- Real programmers' programs never work right the first time. But if you throw them on the machine they can be patched into working in only a few 30-hour debugging sessions.
- Real programmers don't write in Fortran. Fortran is for pipe stress freaks and crystallography weenies.
- Real programmers never work 9 to 5. If any real programmers are around at 9 am, it's because they were up all night.
- Real programmers don't write in BASIC. Actually, no programmers write in BASIC, after the age of 12.
- Real programmers don't document. Documentation is for simps who can't read the listings or the object deck.
- Real programmers don't write in Pascal, or Bliss, or Ada, or any of those pinko computer science languages. Strong typing is for people with weak memories.
- Real programmers know better than the users what they need.
- Real programmers think structured programming is a communist plot.
- Real programmers don't use schedules. Schedules are for manager's toadies. Real programmers like to keep their manager in suspense.
- Real programmers think better when playing adventure.

I found the above at http://www.psych.usyd.edu.au/pdp-11/rp.html while I was perusing some PDP-11 information on the web recently. As I read this “Real Programmers’ Code of Conduct, I busted out laughing in my office in Mickey D’s and I thought of my many friends I finally met in person at this year’s CoCoFEST, so I just had to submit this to the Glenside Color Computer Club Newsletter for publication! It is hilarious as long as you don't take it literally of course. Can anyone relate to the environment these programmers must have been in when programming the monstrous min-frames and main-frames of yesteryear? I would love to hear your stories! Starfleet Out! Qaplah! End Transmission!

Donkey Kong Remixed Review and Walkthrough

by: Richard Goedeken

As a child of the 80s, I've always had a keen interest in all kinds of video games. I played arcade games as a kid, and I recall playing Donkey Kong a few times, though I never tried to master it. When Sock Master originally released his CoCo 3 Donkey Kong port in 2007, I no longer owned any real Tandy hardware on which I could play it, but I started building a meta-emulator system called VGMuseum, which eventually grew to contain over a dozen open source emulators capable of playing thousands of games. In 2012 I set up a VGMuseum system at my workplace, and since then many colleagues have found and played their favorite games from yesteryear.

One colleague in particular (I'll call him EDC), spent lots of time honing his Donkey Kong skills as a kid, and I really enjoyed watched him play arcade Donkey Kong on the VGMuseum. I had never before seen anyone reach the "Pie" level.

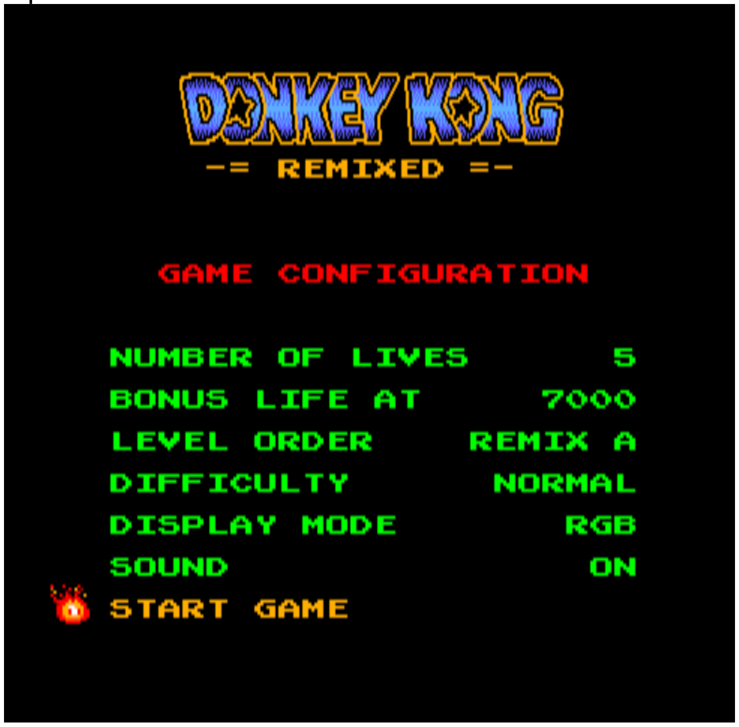
When Donkey Kong Remixed was released this summer, I knew that I had to get it running at the office to get EDC's opinion and see the new levels. I got it running with MESS, and we started playing a few games per day for stress relief and fun. Soon we got hooked. We began a quest to play and defeat each level.

Now, after playing hundreds of games in this quest to complete Donkey Kong Remixed, I cannot overstate my opinion of this game: it's absolutely brilliant. Sock Master's original Donkey Kong port for the CoCo 3 in 2007 is a masterpiece. Not only is it the best arcade-style game ever made for the Color Computer 3, it is the best port of arcade Donkey Kong for any 8-bit machine. All of the DK releases for the Commodore 64 were bad. Even the Colecovision port, which I have played extensively and was considered to be great at the time, pales in comparison to Sock Master's.

But Donkey Kong Remixed takes it to the next level. Not only is it a technical masterpiece, but the new levels show real creativity and balance. However it is not for the tame at heart. It is true to the spirit of Donkey Kong, as seen even in the latest titles from Nintendo. It is incredibly hard and it forces you to practice and learn in order to advance. The arcade Donkey Kong (and Remixed) games are not pure platformers (like the Wii U game, focused only on jumping mechanics and timing), but blend the illusion of chance with an evil machine using coordinated actors to strike you down at any moment. You must use tactics and quick reflexes to make your way through each scene.

As EDC says, this game is rigged. It certainly appears that way at first. But once you learn its ruthless rules and how to circumvent them, you can beat it. In order to defeat this game however, you must be prepared to rise above.

Options



GAME CONFIGURATION

NUMBER OF LIVES	5
BONUS LIFE AT	7000
LEVEL ORDER	REMIX A
DIFFICULTY	NORMAL
DISPLAY MODE	RGB
SOUND	ON
START GAME	

The original Sock Master Donkey Kong allowed the player to select options in several different categories before playing the game, similar to DIP switch settings on an arcade machine. Donkey Kong Remixed has the same categories as the original, but some new options.

You can select either 3, 5, 7, or 9 lives per game, and select the score at which you will be given one extra life. The Level Order can be set to Remix A, Remix B, or Classic. This determines the order in which you will play the scenes. There are more Display Modes (Palette Types) in the Remixed game than the original, including one RGB, two slightly different Composite palettes, and a stylized PMODE 4 (red/blue) option. That last one is a nice touch, and shows Sock Master's attention to detail. The difficulty setting allows for either Easy or Normal play, though in my opinion these should be called Hard and Really Darn Hard instead.

The Scenes

The arcade Donkey Kong game was notable for many reasons. When it was released in 1981, it was the most complex arcade game made. Not only was Donkey

Kong the first game to contain a storyline that is shown graphically as the player completes the levels, but it was also the first game for which the development of the storyline preceded any programming work.

The arcade game contains 4 Scenes, each of which can played at different Levels of difficulty, increasing as the player advances. The Remixed game adds 6 new Scenes to the game. In both the Remix A and Remix B level orders, the player must complete 11 levels in order to see all 10 unique game Scenes.

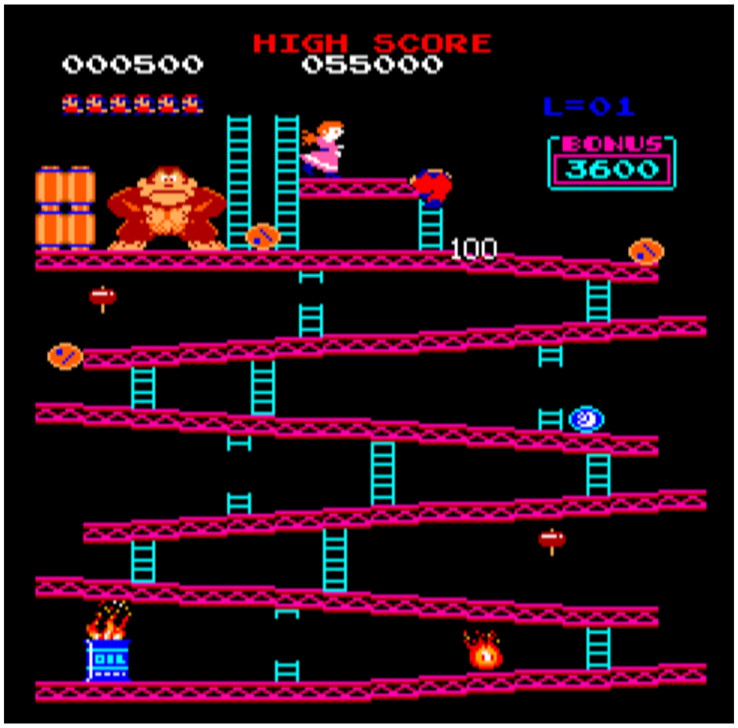
I believe that the Remix A order is the most pure arcade experience, as it nicely blends the new Scenes in with the old ones and introduces the new Scenes gradually. The Remix B order gives you 5 out of the 6 new Scenes at the beginning (when the difficulty is lower), so it's nice for those who wish to see and play the new levels without quite so much difficulty.

In the Level Order lists below, the "L" number before each Scene name gives the difficulty of that level when the game has been set to the Easy mode.

Remix A Level Order
1: L1 Original "Barrels" (tilted red girders)
2: L1 Original "Rivets" (flat blue girders)
3: L2 New "Gumball Machine" (domed girders)
4: L2 Original "Elevators" (bouncing springs)
5: L2 New "Rivets" (flat blue girders)
6: L3 New "Barrels" (reverse tilted red girders)
7: L3 Original "Factory" (pies + conveyor belts)
8: L3 New "Elevators" (center start)
9: L3 New "Rivets and Elevators" (with switch)
10: L4 Original "Barrels"
11: L4 New "Factory" (two flaming oil drums)
12: L4 New "Gumball Machine"
13: L4 Original "Elevators"

Remix B Level Order
1: L1 New "Gumball Machine"
2: L1 New "Elevators"
3: L1 New "Rivets"
4: L2 New "Barrels"
5: L2 New "Factory"
6: L2 Original "Elevators"
7: L2 New "Rivets and Elevators"
8: L3 Original "Barrels"
9: L3 Original "Factory"
10: L3 New "Elevators"
11: L3 Original "Rivets"
12: L4 New "Gumball Machine"
13: L4 New "Factory"
14: L4 Original "Elevators"

Walkthrough



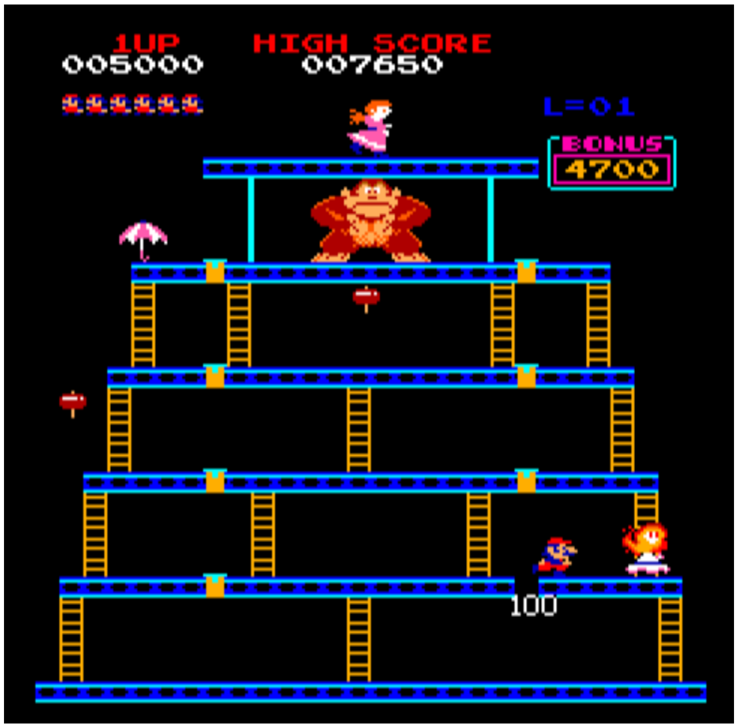
Original "Barrels"

This is the quintessential Donkey Kong level, played and recognized more than any other, since it's the first. But already the viciousness of this game is on full display. Even an experienced player can end up losing lives (sometimes several!) on this level if he lets his guard down. To win this level, your goal is to climb onto the platform at the top upon which Pauline is held captive.

The first lesson to be learned here is the necessity of being quick and precise. There is no time to lose, and all of the levels are easier if you get through them quickly. To do this you must 'hit' onto the ladders by pressing up on the joystick at precisely the right moment in time. If you're just a little too early or too late you'll miss the ladder and waste precious time. The same goes for getting off of the ladder: you must move the joystick to your desired walking direction (left or right) at the right instant to get off the ladder without moving too early and getting stuck or too late and wasting time.

Balancing against the need for speed, you must also use tactical positioning techniques in this level to avoid being clobbered by the barrels. Here's your first hint of the devious nature of this game. Imagine that there is an evil genius controlling the rolling barrels. At each point where a barrel encounters a ladder going down (including broken ones), the hidden genius may tell the barrel to either continue rolling down the platform, or to turn and go down the ladder. In most cases the barrel will just roll on by. But if you are standing or walking towards the ladder with a position and speed such that the barrel will kill you if it turns, then the evil genius will roll a dice and decide whether or not to kill you. It seems that at least 75% of the time, it will. The game is rigged.

In order to strategically counter this behavior, you must not race the barrels past the ladders. Keep an eye on the barrels above, and if you are approaching a ladder at the same time as a rolling barrel above, stop and wait by the side of the ladder. If the barrel goes down, you can jump straight up and let the barrel go underneath you. If it continues on the girder above, you can resume walking. Similarly, you should never stop while climbing up a ladder and wait for a barrel to pass by on the girder overhead. In most cases, it will kill you.



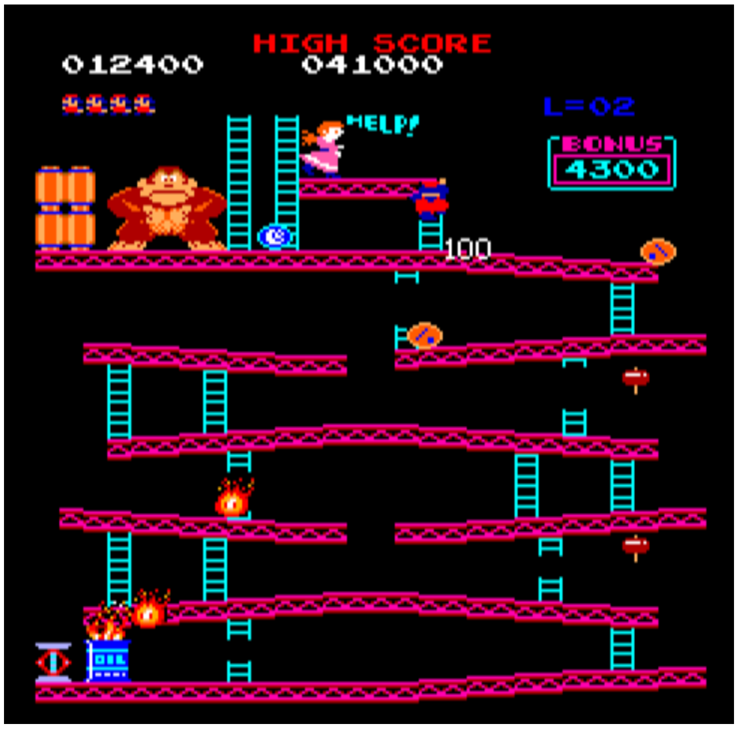
Original "Rivets"

When you win this level, it shows the "victory" scene from the arcade game: Donkey Kong will end up on his head and Mario will be reunited with his lovely Pauline. To do this, you must remove (by walking or jumping over) each of the 8 yellow rivets.

This level is the first to feature the fireballs. In general, they are not very aggressive. They do tend to move towards Mario, but they don't pursue him relentlessly, and often switch directions as though they forgot where they were going. You can take advantage of this by getting relatively close to them to take out the rivets in a desirable order. But don't try to jump over them unless it's an emergency, because they will often switch directions and kill you when you jump them. The fireballs cannot cross over missing rivets, so you can box them in to certain areas of the level for your advantage.

The fireballs appear at random locations on this level, so you must choose your path based on the order in which they appear. One strategy for this scene is to go back and forth, collecting both rivets from each girder before going up. Another effective strategy is to gather all of the rivets in one vertical column, cutting off the side of the level from the center, and then go back down to take out the other side.

When you win this level you will see an extra animation: the girders fall down smoothly to land at the bottom. This animation is a special feature of "DK Remixed" and was not in the arcade game.



New "Gumball Machine"

In the Remix A level order, this is the first new level and almost certainly the hardest one out of the first 11. It is also one of the most brilliantly designed. Take a good look at Mario reaching his goal in this picture, because you're unlikely to see it again any time soon.

Notice the two central gaps between girders in this level: at each gap, the barrels falling down from the sides can either continue in the same direction that they were previously going (left or right), or slow down and switch directions. While waiting on the girder below, you need to know in which direction the barrels from above will go, to take evasive action. Luckily, the game gives you a clue. When a barrel is going to switch directions, it will slow down just a little as it comes to the end of the girder. By watching carefully, you can learn to judge if the barrel will switch direction or not.

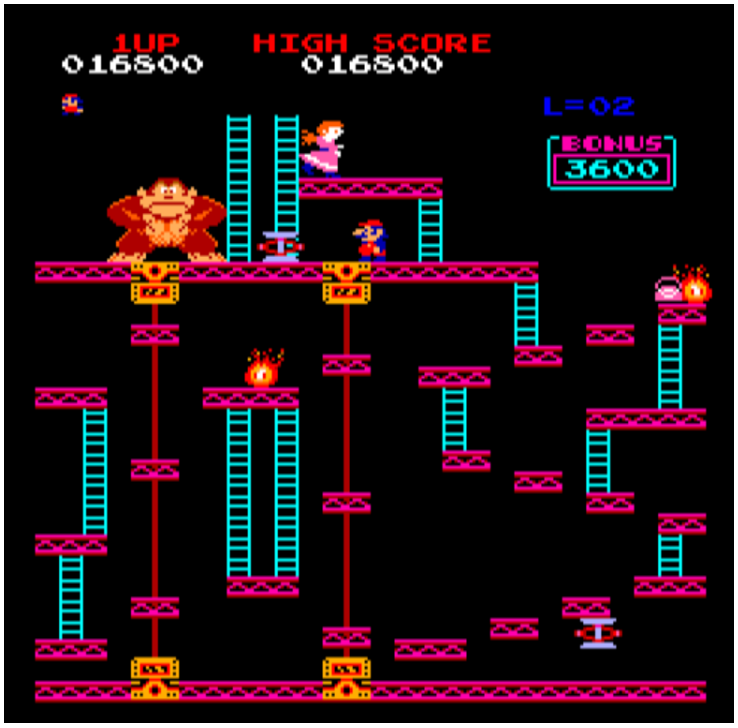
There are two hammers on this level, placed so you can grab them when you're on the girders beneath the central gaps. The hammer can be useful here, but it can also be a double-edged sword, so you must use it carefully. You cannot climb ladders or jump while you have the hammer, so you'll be stuck to one girder for some time after you take it. Also, the hammer will only smash objects which are exactly above or in front of you, in the alternating flashing hammer head squares. If you stand directly underneath a central gap, the barrels coming from above can fall diagonally in between the two hammer head areas, and kill Mario. It's better to stand under one of the girders and face the center. The hammer has another problem as well. When it shuts off, it leaves you helpless, but you can't jump before it shuts off. Learn to estimate the duration of the hammer power so that you can move to safety before it goes away.

You must also jump over the central gaps twice, and they are bigger than you might think, so it's easy to jump too early and miss the other side. Train yourself to delay the jump until you're about to walk off the edge.

Sometimes, after you jump over the first gap, you may be confronted by a wave of many barrels coming at you. If they are spaced far enough apart, you may avoid them by jumping. But if they are spaced irregularly then they will certainly kill you, so don't be afraid to jump back across the gap to the left to miss them before continuing.

Sometimes it's worthwhile to pick up the upper hammer in this level, especially if lots of barrels are in play. If you smash them, you can rack up a lot of points. But you'll also notice that when you have the hammer, they barrels will usually choose to fall away from you (it's rigged). You can use this to your advantage by running to the left side of the girder, so that the barrels will build up on the right side of the level and you'll have an easier time getting up the ladder when the hammer finishes.

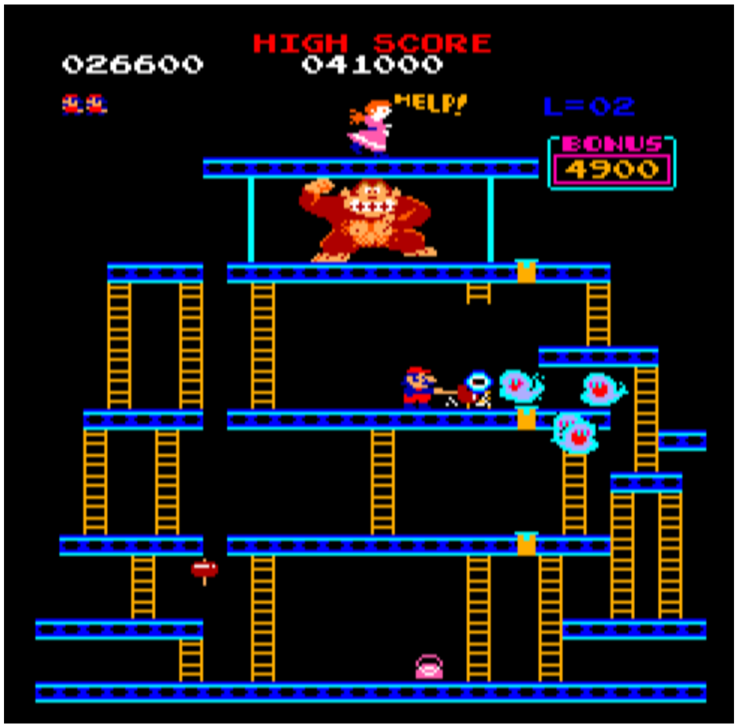
There's one final nasty effect on this level: watch out for that broken ladder just to the right of the top center gap. If Kong has just released a barrel before you jump this gap, it will almost always go down the ladder and kill you.



Original “Elevators”

This is the third level in the arcade game, and it's fairly straightforward to play. Go up one ladder, wait until the fireball starts down the ladder on the right, then jump onto the elevator, ride it up, then jump over to the top platform. Wait until the fireball starts up a ladder and then go down the other one to collect the hat for some points. Then jump onto the second (downward) elevator and then the platform to the right. Jump over one more platform to the right and then wait and time your jumps just right to avoid the springs raining down from above. Make sure to avoid the fireball and time your jumps again to jump up and to the left, and climb the ladders to the top girder with Kong.

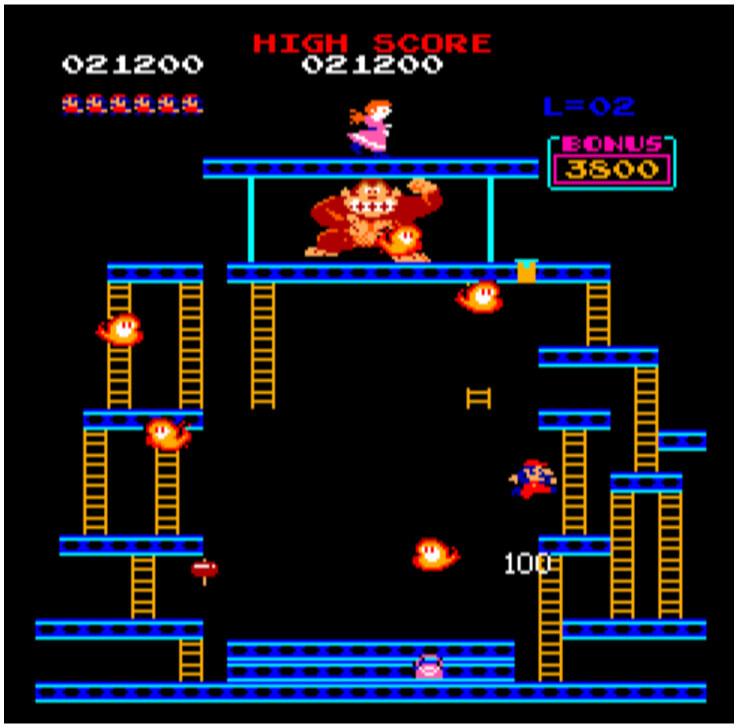
The last part of this level is a little tricky. After climbing the ladder to the top platform, move just a few steps to the left as the springs whiz by. Wait until a spring is flying right over your head and walk to the left. Walk **past** the ladder a bit so that you're positioned just like in the screenshot above. Then wait for the next spring to fly at you, and as it's going overhead, walk to the right and up the ladder. You must hit the ladder on your first try. If you miss the ladder by pressing up too soon, don't try to walk right and hit it again; your only chance is to go back to the safe spot on the left and then wait for another chance.



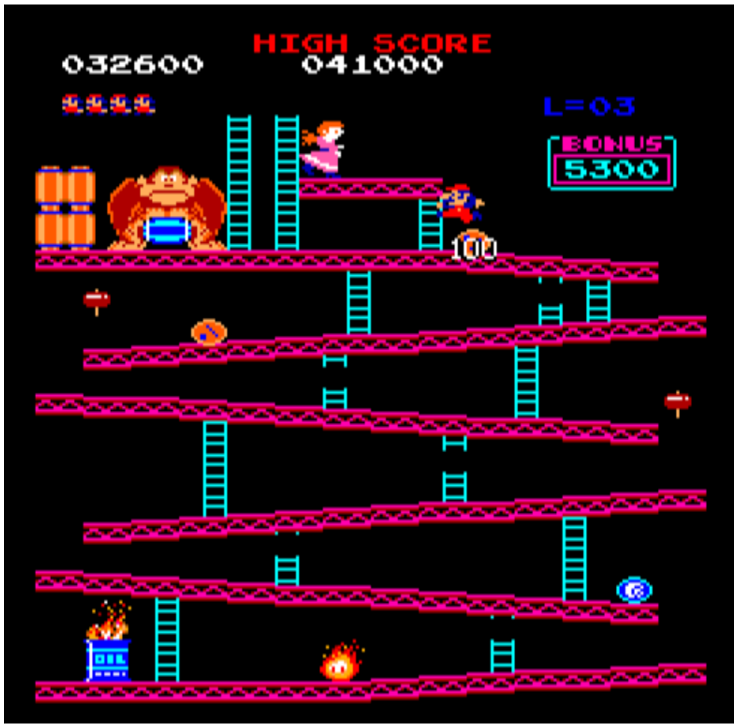
New “Rivets”

The second new level, this one is also quite difficult and requires some strategy to defeat. An earlier mid-July release of DK Remixed had a “feature” whereby Mario could jump from the smallest platform on the right to the one just below it without dying, which made this level very easy. Only the top two rivets had to be removed to defeat Kong and win! Unfortunately Sock Master removed this “feature”, and now it is necessary to get all of the rivets to complete the level.

Start by taking out the 3 rivets in the leftmost column. Then jump onto the middle platform (with the hammer) and try to lure the fireballs up to you. Wait and get the hammer at the right time and then kill as many fireballs as possible. With luck, they will respawn on the left side of the board and you'll be safe from them. Get the two lower rivets on the right. After walking over the second one, the platforms will fall out from underneath you. React quickly, and you can jump back to the right to safety, as shown here:

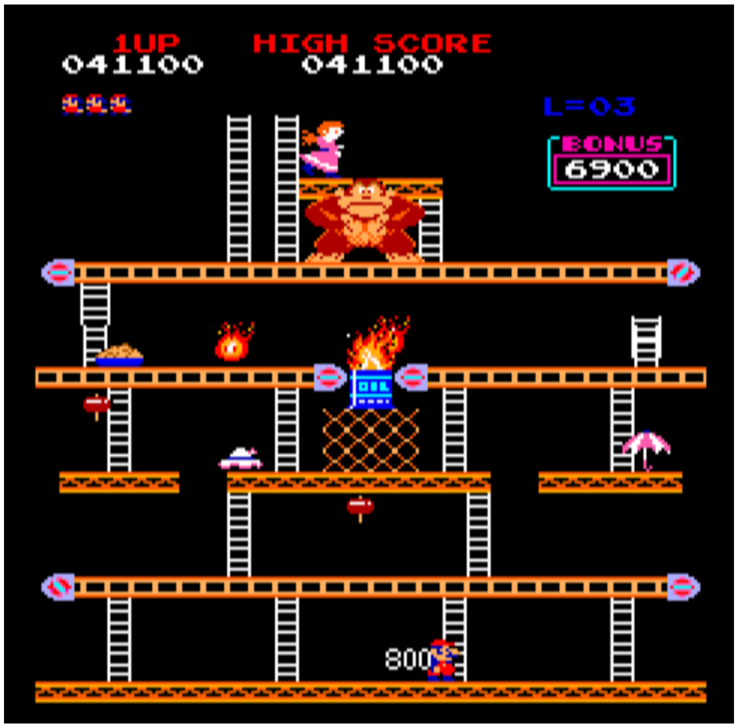


Climb down the ladder to the bottom, run over and collect the purse (without jumping), then jump up on top of the long platforms that you just dropped. From here, jump over to the platform on the right and then go up the ladders to get the final rivet on top.



New “Barrels”

This scene is very similar to the very first one, except the ladders are arranged so that you need to run **down** the girders and then climb up long ladders in order to progress upwards. It plays very similarly to the first level, and you just need to be careful with your position to avoid getting killed by barrels coming down the ladders. Overall it's not too hard.

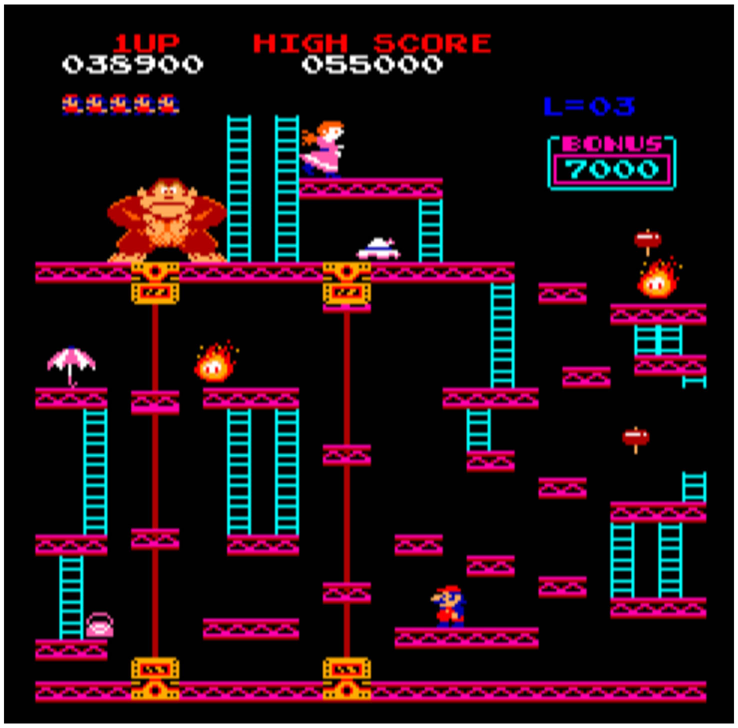


Original “Factory”

Most arcade players know this as the “pie” level, though officially the things that look like pies are actually pans of cement moving through a factory. In the Easy setting of Remix A, this is the first level which will give you 800 points for each item that you collect, so it's good for your score to get a few of them if possible.

The conveyor belts can switch directions, and pies can appear from the ends at any time, so there is a big element of chance when you climb onto the conveyors in this level. It's common to get swarmed by pies and fireballs when you're on the lower conveyor belt. If this happens, you can pick up a lot of points by grabbing the hammer and smashing them. When walking against the direction of travel of the conveyor belt, you will move very slowly. You can gain ground faster by jumping. It's often necessary to jump to reach the ladder if the conveyor belt is going the wrong way. To do this, immediately push up on the joystick when you're in the air jumping past the ladder, and let the conveyor carry you to the ladder. You'll start climbing as you pass it.

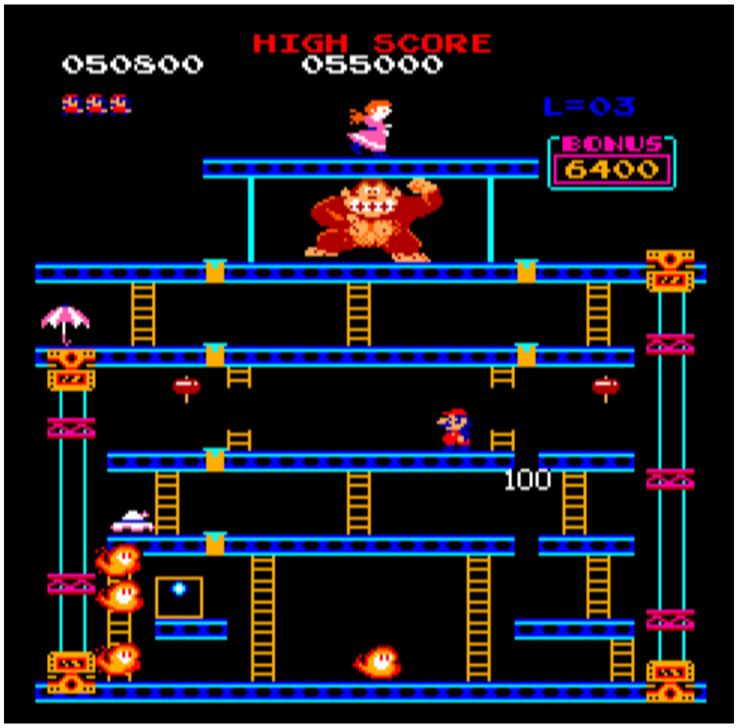
Once you reach the top of this level you're pretty much home free. You have to go up to Pauline's platform from the ladder on the right, but you can get there from either side of Kong. Once you reach the top, just let the conveyor push you to one side (it won't throw you off) and then wait until it switches directions. Then run up close to Kong (but don't touch him or you'll die), and wait until the conveyor pulls you to Pauline's ladder, then run up it to win.



New “Elevators”

This level is very similar to the original Elevators level, except you start on the big platform in the middle instead of on the left. Begin by going all the way to the left and climbing up the ladders, then going to the right. You must jump onto the second elevator from the top platform, otherwise you probably won't be able to make it to the upper small platform to the right. The timing of the jumps in this level is a little tighter than the original level, and you have to be more attentive to the lateral position of the springs, because they vary more in their left/right position than in the original level.

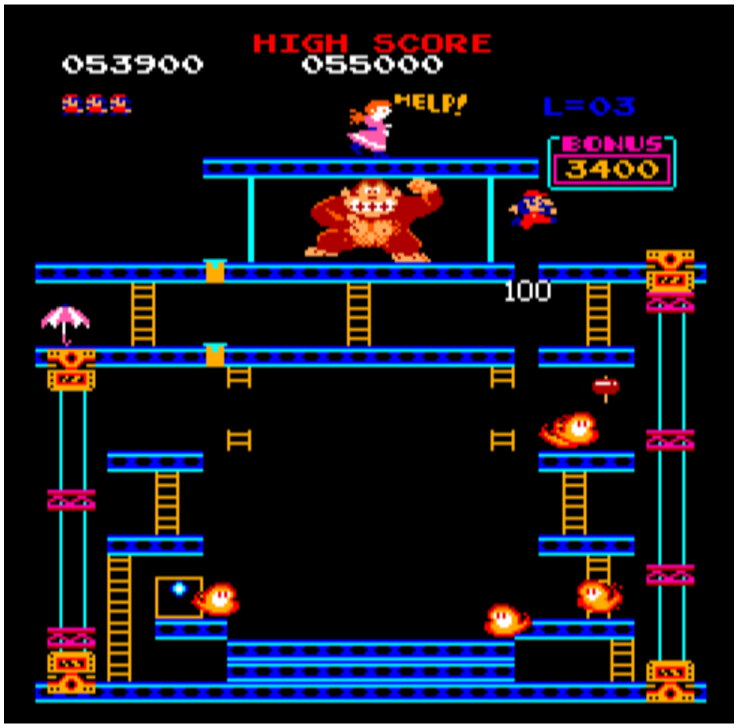
You need to complete the top part of this level in the same way as the original; run to the left, **past** the ladder to Pauline, and wait for another spring to pass overhead before running right, then go up the ladder to win.



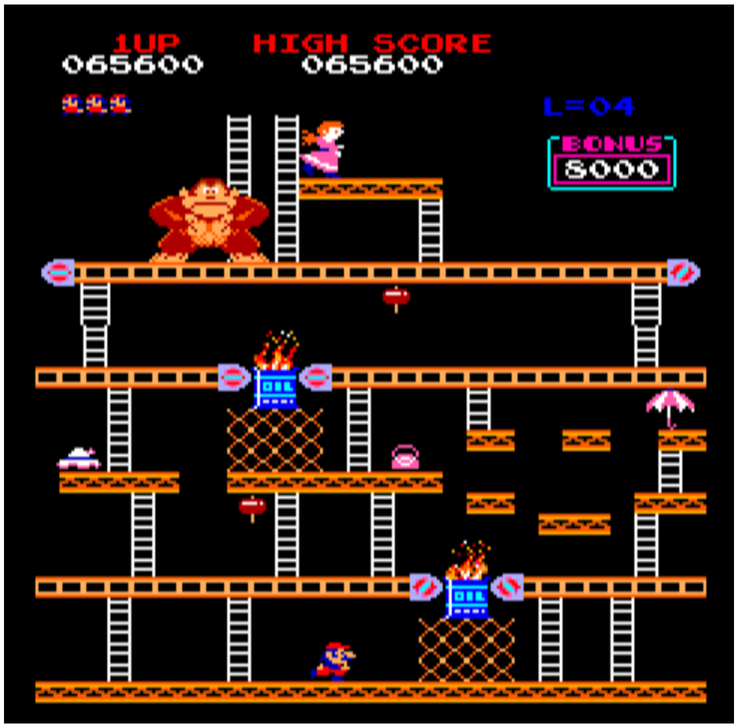
New “Rivets and Elevators”

This level is kind of crazy. There are so many different things going on here. Start by running to the right, snatch the purse, and get the two bottom rivets on the right. On the middle platform it's good to use one or both of the hammers to take out the fireballs, which conspire to make this a difficult level. You can also walk over the leftmost rivet on top while you have the hammer to drop this platform and kill the fireballs underneath.

Next, go down the small ladder on the left and take out the final rivet of the bottom four to drop these two platforms down. As on the first new Rivets level, you'll have to get the rivet and then quickly jump back to the left as the platforms start falling, to avoid death. Go down the ladder to the very bottom and then jump up on top of the two platforms that you just dropped. Jump up to that switch on the left. (Yep, it's a switch. Brilliant!) The elevators will change directions. Run over to the right (avoiding the fireballs) and ride up the elevator to the top section of this level. You must remember to start here by first getting **both** rivets on the right, otherwise you will not be able to finish the level.



After getting the top right rivet as shown above, go down, then to the left (again avoiding the fireballs), and get the last two rivets on the left to win. Winning this Scene involves a lot of luck regarding where the fireballs appear and how they move, so don't get too discouraged. It's not quite as hard as it looks, but it's no cakewalk either.



New “Factory”

The last new level is also very creative. I've only played it a few times, but I've had good luck by running up to the floating platforms on the right side. The fireballs cannot follow you there, so you have some natural protection. Feel free to grab the hammer on the penultimate conveyor belt to rack up some points and get rid of the pies. Once you get to the top conveyor you can use the same technique as in the original Factory level to easily reach the platform with Pauline.

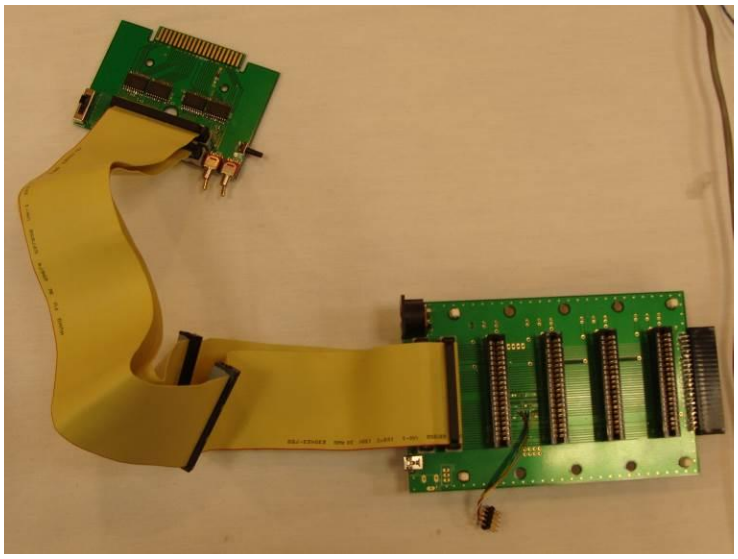
Conclusion

As it should be clear by now, Donkey Kong Remixed is not an easy game for a casual gamer. If you're looking easy wins, you're likely to be overly frustrated by this one. It is not nice. It forces you to confront your inner limitations, learn, and grow in order to advance. But it is extremely satisfying when you make it to a new level. If you yearn to experience a brand new CoCo gaming masterpiece, inspired by one of the greatest arcade games from the dawn of the gaming era, Donkey Kong Remixed cannot be missed. If you have a few (dozen) hours to invest and pretty good reflexes, you might even complete the quest and win all 10 of the Donkey Kong Remixed scenes.

New Multi-Pack Prototype
by John Mark Mobley

Jim Brain of Retro Innovations has designed a new multi-pack prototype and brought it to the Vintage Computer Festival Midwest (VCFMW). Neil Blanchard and Jim O’Keefe tested the prototype. It worked about 95% of the way. The one issue they found may be an issue with other hardware. The new multi-pack prototype is controlled by configurable logic such as a Complex Programmable Logic Device (CPLD). So making changes to the logic may not require a board redesign.

Jim Brain (right) with Retro Innovations


New Multi-Pack Prototype

(Left to right) Jim O’Keefe, Neil Blanchard, and Walter Miraglia

Overflow Logic
by John Mark Mobley

During the 1980s many cars had mechanical odometers. They would roll over to zero after 100,000 miles. So they would count up to 99999.9 miles and if you drove 0.1 mile more it would read 00000.0. This is an example of overflow. Computers do something similar with overflow logic.

If an 8-bit add overflow occurs, then you can likely just do a 16-bit add in place of an 8-bit add, and you will get the correct answer. Table 1 shows how many bits are required to get the necessary precision. This is not packed BCD, it is just binary.

Often you just want to compare two numbers and branch if A>B. In this case you do not need to store the results, so you can ignore the overflow condition and move on.

Computers can add signed or unsigned integers using the same add instruction. When the add is done, the microprocessor will adjust flags in the Condition Code Register (CC or CCR). You, the programmer, need only pay attention to the flags that are related to the type of math you are doing, either signed or unsigned.

Bytes	Bits	Signed Base 10 Digits	Unsigned Base 10 Digits
1	8	2	2
2	16	4	4
3	24	6	7
4	32	9	9
5	40	11	12
6	48	14	14
7	56	16	16
8	64	18	19
16	128	38	38
32	256	76	77
64	512	153	154

Table 1: Bits to Digits (Ex: 8 bits equal about 2 digits)

Signed Overflow Logic.

The condition code register has an overflow flag or V flag that is used to indicate a signed overflow. The condition code register has a negative flag or N that is used to indicate that a signed number is negative. If the most significant bit of a signed number is set, then the number is negative.

The yellow highlighted areas below represent a 3-bit signed overflow.

Signed 3-bit Add (Carry In = 0)

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

Table 2

C Code	BASIC	6809 ASM	Logic Gate
&	AND	AND	And Gate
|	OR	OR	Or Gate
!	NOT	COM	Inverter

Table 3: C, BASIC, ASM and Logic Gate Notation

Add B to A
R=A+B;
8-bit Overflow = (A7 & B7 & !R7) | (!A7 & !B7 & R7)
Overflow = (SignOfA & SignOfB & !SignOfResults) | (!SignOfA & !SignOfB & SignOfResults)
Overflow = ((A < 0) & (B < 0) & (R >= 0)) | ((A >= 0) & (B >= 0) & (R < 0));

If the following statements are true, then the overflow flag is cleared (V=0). A positive number plus a positive number is a positive number. Pos + Pos = Pos. A negative number plus a negative number is a negative number. Neg + Neg = Neg.

Example 8-bit signed add overflows to a 16-bit result:
Dec: 127 + 127 = 254
Hex: 0x7F + 0x7F = 0x00FE

Example 8-bit signed add overflows to a 16-bit result:
Dec: (-128) + (-128) = (-256)
Hex: 0x80 + 0x80 = 0xFF00

Signed 3-bit Subtract Col-Row (Carry In = 0)

	-4	-3	-2	-1	0	1	2	3
-4	0	1	2	3	4	5	6	7
-3	-1	0	1	2	3	4	5	6
-2	-2	-1	0	1	2	3	4	5
-1	-3	-2	-1	0	1	2	3	4
0	-4	-3	-2	-1	0	1	2	3
1	-5	-4	-3	-2	-1	0	1	2
2	-6	-5	-4	-3	-2	-1	0	1
3	-7	-6	-5	-4	-3	-2	-1	0

Table 4

Subtract B from A
R = A – B;
Overflow = (A7 & !B7 & !R7) | (!A7 & B7 & R7)
Overflow = (SignOfA & !SignOfB & !SignOfResults) | (!SignOfA & SignOfB & SignOfResults)
Overflow = ((A < 0) & (B >= 0) & (R >= 0)) | ((A >= 0) & (B < 0) & (R < 0));

If the following statements are true, then the overflow flag is cleared (V=0). A positive number minus a negative number is a positive number. Pos - Neg = Pos. A negative number minus a positive number is a negative number. Neg - Pos = Neg.

Example 8-bit signed sub overflows to a 16-bit result:
Dec: 127 – (-128) = 255
Hex: 0x7F - 0x80 = 0x00FF

Example 8-bit signed sub overflows to a 16-bit result:
Dec: (-128) – 127 = -255
Hex: 0x80 - 0x7F = 0xFF01

Unsigned Overflow Logic.

The condition code register has a Carry flag or C flag that is used to indicate an unsigned overflow.

The yellow highlighted areas below represent a 3-bit unsigned overflow.

Unsigned 3-bit Add (Carry In = 0)

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	8
2	2	3	4	5	6	7	8	9
3	3	4	5	6	7	8	9	10
4	4	5	6	7	8	9	10	11
5	5	6	7	8	9	10	11	12
6	6	7	8	9	10	11	12	13
7	7	8	9	10	11	12	13	14

Table 5

Add B to A
If the carry flag is set (C=1) after the add/adc is complete, then the result has overflowed.

Example 8-bit unsigned add overflows to a 16-bit result:
Dec: 255 + 255 = 510
Hex: 0xFF + 0xFF = 0x01FE

The yellow highlighted areas below represent a 3-bit unsigned overflow or signed result.

The orange highlighted areas below represent a 3-bit unsigned overflow or signed overflow.

Unsigned 3-bit Subtract Col-Row (Carry In = 0)

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	-1	0	1	2	3	4	5	6
2	-2	-1	0	1	2	3	4	5
3	-3	-2	-1	0	1	2	3	4
4	-4	-3	-2	-1	0	1	2	3
5	-5	-4	-3	-2	-1	0	1	2
6	-6	-5	-4	-3	-2	-1	0	1
7	-7	-6	-5	-4	-3	-2	-1	0

Table 6

Subtract B for A
If the carry flag is set (C=1) after a sub/sbc, then the result is a signed negative number. This is technically an unsigned overflow.

If the carry flag is set and the negative flag is clear (C=1 and N=0) after a sub/sbc, then a signed overflow has occurred.

CC Flags	Results
C=0	Unsigned results
C=1 and N=1	Unsigned overflow or signed results
C=1 and N=0	Unsigned overflow and signed overflow

Table 7

So subtracting unsigned numbers can give a signed result.

8-bit Example 1 (unsigned results):
Dec: 255 - 0 = 255
Hex: 0xFF - 0x00 = 0xFF (C=0)

8-bit Example 2 (unsigned overflow or signed results):
Dec: 1 - 2 = (-1)
Hex: 0x01 - 0x02 = 0xFF (C=1, N=1)

8-bit Example 3 (unsigned overflow and signed 8-bit overflow):
Dec: 0 - 255 = -255
8-bit Hex: 0x00 - 0xFF = 0x01 (C=1, N=0)
16-bit Hex: 0x0000 – 0x00FF = 0xFF01 (C=1, N=1)

If you are expecting signed results, then you should probably use signed math to begin with. That way there is no question on how to view the results.

Signed Overflow Logic and the Neg Command

Many operations can set the overflow flag in the condition code register.

Example:

```
lda #-128
nega
bvs OverflowErrorHandler01
```

A signed byte can only go from -128 ($80) to +127 ($7F). The negative of -128 should be +128 but that answer cannot be stored in a signed byte. So it is an overflow. The answer it gives is -128 ($80). The correct answer is +128 ($0080). You can trap this error using the BVS (Branch Overflow Set) command.

Vintage Computer Festival Midwest (VCFMW)
by John Mark Mobley

I got DriveWire running on my Linux Mint MATE Laptop and demonstrated Donkey Kong Remix. I got some positive comments from people saying that they liked this version of the game.

John Mark Mobley Demo-ing Donkey Kong Remix

Assembly 32-bit Compare Subroutine
by John Mark Mobley

Most of the branch instructions rely on flags being set in the condition codes in order to determine if the branch is to occur. It is like an "IF" statement in BASIC.

Ex: BASIC

```
 100 A=128
 110 IF CARRY=1 THEN 130.
 120 A=0
 130 PRINT A
 140 RETURN
```

Ex Assembly 1

```
 PrintValue   lda #128       ; A=128
              bcs PV130      ; Branch Carry Set
              lda #0         ; A=0
 PV130        jsr PrintUnsignedDec8Bit
              rts            ; Return
```

Ex Assembly 2

```
 PrintValue   lda #128       ; A=128
              bcs >          ; Branch Carry Set
              lda #0         ; A=0
 !            jsr PrintUnsignedDec8Bit
              rts            ; Return
```

In the Ex Assembly 2 I use ">" and "!" in place of the address label "PV130". ">" looks forward to the first "!" and "<" looks backward for the first "!". This notation works in the Lost Wizard Assembler "LWASM". "LWASM" is part of Lost Wizard Tools or "LWTools" See Table 1 for a list of branches.

Mnemonic	Description
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Equal
BGE	Branch on Greater than or Equal to Zero
BGT	Branch on Greater
BHI	Branch if Higher
BHS	Branch if Higher or Same
BLE	Branch on Less than or Equal to Zero
BLO	Branch on Lower
BLS	Branch on Lower or Same
BLT	Branch on Less than Zero
BMI	Branch on Minus
BNE	Branch Not Equal
BPL	Branch on Plus
BRA	Branch Always
BRN	Branch Never
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set

Table 1: Branch Mnemonics and Descriptions

The condition codes are set, cleared or unaffected by the various preceding instructions. Table 2 shows what flag settings are required to allow the branch to be taken as opposed to skipped.

Mnemonic	Equation	Type
BCC, BHS	C=0	Simple, Unsigned
BCS, BLO	C=1	Simple, Unsigned
BEQ	Z=1	Simple, Unsigned, Signed
BGE	N xor V = 0	Signed
BGT	Z or (N xor V) = 0	Signed
BHI	C or Z = 0	Unsigned
BLE	Z or (N xor V) = 1	Signed
BLS	C or Z = 1	Unsigned
BLT	N xor V = 1	Signed
BMI	N = 1	Signed
BNE	Z = 0	Simple, Unsigned, Signed
BPL	N = 0	Simple
BRA	1	Unary
BRN	0	Unary
BVC	V = 0	Simple
BVS	V = 1	Simple

Table 2: Branch Instruction Summary (8-bit Offset)

Mnemonic	Complement
BCC, BHS	BCS, BLO
BCS, BLO	BCC, BHS
BEQ	BNE
BGE	BLT
BGT	BLE
BHI	BLS
BLE	BGT
BLS	BHI
BLT	BGE
BMI	BPL
BNE	BEQ
BPL	BMI
BRA	BRN
BRN	BRA
BVC	BVS
BVS	BVC

Table 3: Branch and its Complement

The compare instructions set the condition codes needed to select any branch. The compare instructions in the 6809 can do an 8-bit or 16-bit compare depending on whether the register you are comparing is an 8-bit or 16-bit register. The compare instructions can compare two unsigned numbers or two signed numbers automatically, meaning you do not have to tell it in advance what number system you are using. However, you must choose the correct branch instruction to match either unsigned or signed math. See Table 4. An 8-bit or 16-bit compare with zero can usually be done without doing a compare with zero. See the lower half of Table 4. The Test instructions only set/clear the N, V and Z flags, which is adequate to do an 8-bit compare with zero. Because the C flag is not cleared by the test instruction, some unsigned branches will not work correctly, but the lower half of Table 4 gives some ideas on how to get around this.

BASIC	C Code	6909 ASM Unsigned	6809 ASM Signed
>	>	BHI	BGT
>=	>=	BHS, BCC	BGE
=	==	BEQ	BEQ
<=	<=	BLS	BLE
<	<	BLO, BCS	BLT
<>	!=	BNE	BNE
> 0	> 0	BNE	BGT (See Note 1)
>= 0	>= 0	BRA !!!	BPL
= 0	== 0	BEQ	BEQ
<= 0	<= 0	BEQ	BLE (See Note 1)
< 0	< 0	BRN !!!	BMI
<> 0	!= 0	BNE	BNE
Note 1: Make sure V=0 The following commands clear the V flag: ANDA, ANDB, ANDCC #$FD, BITA, BITB, CLR, CLRA, CLRB, CMPA #0, CMPB #0, CMPD #0, CMPS #0, CMPU #0, CMPX #0, CMPY #0, COM, COMA, COMB, DAA, EORA, EORB, LDA, LDB, LDD, LDS, LDU, LDX, LDY, ORA, ORB, SEX, STA, STB, STD, STS, STU, STX, STY, SUBA #0, SUBB #0, SUBD #0, TST, TSTA, TSTB			

Table 4: Unsigned and Signed Branches

Example 8-bit compare:

```
        lda NS8
        cmpa #100
        bgt Skip101
```

Example 8-bit compare with zero

```
        lda NS8
        bgt Skip101
```

Example register-less 8-bit compare with zero

```
        tst NS8
        bgt Skip101
```

Example 16-bit compare

```
        ldx NS16
        cmpx #-1       ; #-1= #$FFFF
        bgt Skip102
```

Example 16-bit compare with zero

```
        ldx NS16       ; if X2>0 goto Skip102
        bgt Skip102    ;
```

Example 32-bit compare

```
        ldx NS32       ; Signed 32-bit number
        ldu XS32       ; Signed 32-bit number
        jsr Cmp32      ; IF [X]>[U] GOTO Skip103
        bgt Skip103    ; “
```

Example 32-bit compare with zero

```
        ldx NS32       ; Signed 32-bit number
        jsr Tst32      ; IF [X]>0 GOTO Skip103
        bgt Skip103    ; “
```

Example 32-bit test for plus or minus

```
        tst NS32       ; Signed 32-bit number
        bpl Skip103    ; IF NS32 >= 0
```

If you want to do a 32-bit compare, you can do a subtract instruction, and three subtract with carry instructions. At this point the condition codes are set for Carry, Negative, and Overflow, but the Zero Flag is only set for the most significant 8-bits of the results. Now you can “or” all the bytes of results together and the Zero Flag will be set to a 32-bit test for zero, but doing so may change the value of the Overflow Flag, and the Negative Flag. So you need to save the other flags while you change the zero flag. See Listing 1 for an example of how this can be done.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
E	F	H	I	N	Z	V	C

Table 5 Condition Codes

Condition Code Flags
E =Entire Flag
F =FIRQ Mask
H = Half Carry
I = IRQ Mask
N = Negative Flag
Z = Zero Flag
V = Overflow Flag
C = Carry Flag

Example branch if Half Carry Set

```
        tfr cc,b           ; Transfer Condition Code to B
        bitb #%00100000    ; Test Half Carry Flag
        bne Skip104        ; Branch if Half Carry Set
```

The BRN (Branch Never) and LBRN (Long Branch Never) are another way to do a no-operation (nop). You can use them as code space fillers, you can use them to erase branches, and you can use them in delay loops.

BRA (Branch Always) and BRN (Branch Never) can be used to debug code. They can also be used in self-modifying code.

I once had a program that cleared the “A” accumulator with a CLRA instruction and then did a BCS (Branch Carry Set) but the branch never worked. Then I figured out that the CLRA instruction cleared the “A” accumulator and the Carry Flag, so I changed the instruction from CLRA to LDA #0 and my code started working. Such a small change in my code made a big difference in the outcome. If you work with it every day you will likely develop a knack for it. Otherwise you can rely on a lookup table of instructions and affected condition codes.

Listing 1:

```
* Name: math32.asm, math32.bin, MATH32.BIN
* Purpose: To write a compare routine to handle 32-bit numbers
* By: John Mark Mobley and William Astle
* Date: 07-21-2015
* Run Environment: Tandy Color Computer 1
* Emulator: XRoar
* RAM: 16k
* ROM: Disk Extended Color BASIC
* Development Environment: IBM Compatible PC, Linux, LWTOOLS, and ToolShed
* Build Script:
*   #!/bin/bash
*   lwasm math32.asm -b --list=math32.lst -omath32.bin
*   decb copy -2 -b -r math32.bin cocotest.dsk,MATH32.BIN
*   decb dir cocotest.dsk:0
*   makewav -r -c -nMATH32 -2 -b -k -omath32.cas math32.bin
*
        pragma cescapes
        org $3A00

;Reserve variable space in RAM
N1 rmb 4
N2 rmb 4

*---|----1----|----2----|----3----|----4----|----5----|----6----|----7----|----8
*
* Start... Main Control Section
*
Start   lda  #0         Print to screen
        sta  $006F      "
        ldx #N1         ; N1 = #$00000000
        ldd #$0000      ; "
        std ,x          ; "
        ldd #$0000      ; "
        std 2,x         ; "
        ldu #N2         ; N2 = #$00000000
        ldd #$0000      ; "
        std ,u          ; "
        ldd #$0000      ; "
        std 2,u         ; "
        jsr Cmp32
        rts

*---|----1----|----2----|----3----|----4----|----5----|----6----|----7----|----8
*
* Cmp32... 32 bit compare [x]-[u].
* Compare the 32 bit word pointed to by x from the 32 bit word pointed to by u.
* Works for signed and unsigned numbers
* Works with two unsigned numbers by setting the Condition Code (CC) for
* bhi, bhs (bcc), beq, bls, blo (bcs), bne
* or two signed numbers by setting the ccr for
* bgt, bge, beq, ble, blt, bne, bpl, bmi, bvc, bvs
* The only trick is preserving the value of CC.N and CC.V while adjusting CC.Z
* This function is reentrant tolerant (you may call it from an interrupt)
* Return result in CC (Flags affected: N, Z, V and C).
* Z: Set if result = 0x00000000 (32-bit zero)
* V: Set if two's complement overflow occurs as a result of operation
* C: Set if result needs a borrow (for unsigned numbers it is set if [x]<[u])
* N: Set if results.bit31 is set (set if the two's complement result is
*    negative)
*
*          Unsigned vs. Signed
* +-------+--------+----------+--------+
* |       |        |   ASM    |  ASM   |
* | BASIC | C Code | Unsigned | Signed |
* +-------+--------+----------+--------+
* |  >    |   >    | bhi      |  bgt   |
* |  >=   |   >=   | bhs, bcc |  bge   |
* |  =    |   ==   | beq      |  beq   |
* |  <=   |   <=   | bls      |  ble   |
* |  <    |   <    | blo, bcs |  blt   |
* |  <>   |   !=   | bne      |  bne   |
* +-------+--------+----------+--------+
*
*                 Branch Instruction Summary (8-bit offset)
* +----------+---------------------+--------------------------+------------+
* | Mnemonic |      Equation       |           Type           | Complement |
* +----------+---------------------+--------------------------+------------+
* | bcc, bhs |       C = 0         |     Simple, Unsigned     |  bcs, blo  |
* | bcs, blo |       C = 1         |     Simple, Unsigned     |  bcc, bhs  |
* |    beq   |       Z = 1         | Simple, Unsigned, Signed |    bne     |
* |    bge   |    N xor V = 0      |          Signed          |    blt     |
* |    bgt   | Z or (N xor V) = 0  |          Signed          |    ble     |
* |    bhi   |     C or Z = 0      |         Unsigned         |    bls     |
* |    ble   | Z or (N xor V) = 1  |          Signed          |    bgt     |
* |    bls   |     C or Z = 1      |         Unsigned         |    bhi     |
* |    blt   |    N xor V = 1      |          Signed          |    bge     |
* |    bmi   |       N = 1         |          Simple          |    bpl     |
* |    bne   |       Z = 0         | Simple, Unsigned, Signed |    beq     |
* |    bpl   |       N = 0         |          Simple          |    bmi     |
* |    bra   |         1           |           Unary          |    brn     |
* |    brn   |         0           |           Unary          |    bra     |
* |    bvc   |       V = 0         |          Simple          |    bvs     |
* |    bvs   |       V = 1         |          Simple          |    bvc     |
* +----------+---------------------+--------------------------+------------+
*
* Example:
*        ldx #N1        ; signed 32-bit value
*        ldu #N2        ; signed 32-bit value
*        jsr Cmp32      ; if N1 >= N2 then goto Skip3
*        bge Skip3      ; "
*
Cmp32   pshs d          ;
        ldd 2,x         ; subtract least significant words
        subd 2,u        ; "
        pshs d          ; save results for 24-bit zero test (on the stack)
        ldd ,x          ; subtract most significant words
        sbcb 1,u        ; "
        sbca ,u         ; "
        tfr cc,a        ; accumulator a = cc (E,F,H,I,N,Z,V,C)
        orb ,s+         ; perform a 24-bit zero test
        orb ,s+         ; "
        beq >           ; branch to not clear the Z flag
        anda #%11111011 ; clear Z flag in accumulator a
```

```
!       tfr a,cc        ; cc = accumulator a
        puls d          ;
        rts             ;

*---|----1----|----2----|----3----|----4----|----5----|----6----|----7----|----8
*
* Tst32... 32 bit test of [x].
* Test the 32 bit word pointed to by x.
* Works for signed and unsigned numbers
* Works with one unsigned number by setting the Condition Code (CC) for
* beq, bne
* or one signed numbers by setting the ccr for
* bgt, bge, beq, ble, blt, bne, bpl, bmi
* The only trick is preserving the value of CC.N and CC.V while adjusting CC.Z
* This function is reentrant tolerant (you may call it from an interrupt)
* Return result in CC (Flags affected: N, Z, and V).
* Z: Set if result = 0x00000000 (32-bit zero)
* V: Cleared
* N: Set if bit31 is set (set if the two's complement result is negative)
*
*          Unsigned vs. Signed
* +-------+--------+----------+--------+
* |       |        |   ASM    |  ASM   |
* | BASIC | C Code | Unsigned | Signed |
* +-------+--------+----------+--------+
* |  > 0  |  > 0   | bne      |  bgt   |
* |  >= 0 |  >= 0  | bra !!!  |  bpl   |
* |  = 0  |  == 0  | beq      |  beq   |
* |  <= 0 |  <= 0  | beq      |  ble   |
* |  < 0  |  < 0   | brn !!!  |  bmi   |
* |  <> 0 |  != 0  | bne      |  bne   |
* +-------+--------+----------+--------+
*
* Example:
*        ldx #N1        ; signed 32-bit value
*        jsr Tst32      ; if N1 > 0 then goto Skip3
*        bgt Skip3      ; "
*
Tst32   pshs d          ;
        tst ,x          ; Set/clear N, V, and Z
        tfr cc,b        ; accumulator b = cc (E,F,H,I,N,Z,V,C)
        lda 1,x         ; perform a 24-bit zero test
        ora 2,x         ; "
        ora 3,x         ; "
        beq >           ; branch to not clear the Z flag
        andb #%11111011 ; clear the Z flag in accumulator b
!       tfr b,cc        ; cc = accumulator b
        puls d          ;
        rts

        end Start
```

See you next year: April 23 & 24, 2016

HERON POINT CONVENTION CENTER
645 West North Avenue
Lombard, IL 60148
645 Building on the lower level
Saturday Setup Time 7 am,
Saturday FEST Time 9 am to 5 pm,
Saturday Supper Time 5 pm to 7 pm,
Saturday Evening Social Time 7 pm to 11:30 pm,
Sunday Worship Service Time 8 am,
Sunday FEST Time 9 am to 3 pm,
Sunday Close-Up Time 3 pm to 5 pm

Calendar of Events
by
John Mark Mobley and Salvador Garcia

Glenside Color Computer Club, Inc. Business Meetings
Thursday, October 8, 2015
Thursday, November 12, 2015
Thursday, December, 10, 2015
7:00 PM to 10:00 PM CDT
Schaumburg Public Library
130 South Roselle Road
Schaumburg, IL, USA
Skype access available via: john.mark.mobley

Saturday & Sunday Sept 26-27, 2015
Retropalooza
Arlington, TX USA
Link:
https://www.facebook.com/pages/Retropalooza/171779462982186

Saturday and Sunday Oct 17-18, 2015
Portland Retro Gaming Expo
Portland, OR, USA
Link: http://www.retrogamingexpo.com/

Friday, Saturday and Sunday Nov 13-15, 2015
Syntax DemoParty
Melbourne, Australia
Link: http://www.syntaxparty.org/

Friday, Saturday and Sunday Dec 4-6, 2015
World of Commodore 2015
Toronto, Ontario, Canada
Link: http://www.tpug.ca/category/woc/

Be sure to visit our Website to see up to date information on upcoming events. http://glensideccc.com

The 25th Annual "Last" Chicago CoCoFEST!

2016 CoCoFEST!
Proudly Presented By
The Glenside Color Computer Club
Fairfield Inn, Lombard, Illinois

Here are the 5 "W's"
WHO? 1) Glenside Color Computer Club, Inc.
PRESENTS
WHAT? 2) The 25th Annual "Last" Chicago CoCoFEST!
WHEN? 3) April 23 & 24, 2016
(Sat. 9 am-5 pm; 8 pm-midnight - Sun. 9 am-3 pm)
WHERE? 4) Fairfield Inn & Suites Lombard
645 West North Avenue
Lombard, IL 60148
(Near the intersection of IL-355 and North Avenue)
(Same location as 2015!)
Overnight room rate:
$89.00 plus 11% tax ($98.79 Total)
Call 1-630-629-1500 for reservations.
You must ask for the Glenside "CoCoFEST!" rate.
>>> YOU MUST REGISTER UNDER "CoCoFEST!" <<<
>>> TO GET THIS RATE <<<

WHY? 5)
A. To provide vendor support to the CoCo Community
B. To provide Community support for our CoCo Vendors
C. To provide educational support to new users.
D. TO HAVE AN OUTRAGEOUSLY GOOD TIME!!!

And now, the "H" word.
HOW MUCH? All Attendees - General Admission
Both days: $10.00 1st - $5.00 2nd & more
Sunday Only: $5.00 1st - $5.00 2nd & more
********* Children 12 and under - FREE *********

For further information, general or exhibitor, contact:

Tony Podraza, GCCCI	**Robert Swoger, GCCCI**
847-428-3576, VOICE	**224-236-5194, VOICE**
tonypodraza@gmail.com	**rswoger@aol.com**

Please note the new starting times, 9 AM, not 10 AM.