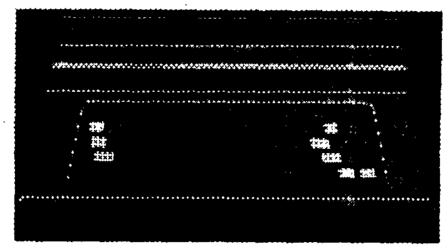
September/October 1997 Vol. 5 Number 2 $4.00US (North America)

the world of 68' micros

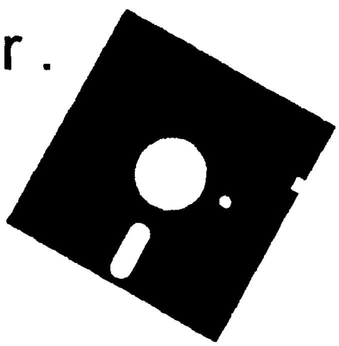
Support for Motorola based computer systems and microcontrollers, and the OS-9 operating system

CoCo or MM/1 RGB Monitor Dead? Look to SOURCES for another!



CONTENTS

New utilities to transfer CoCo3 Emulator .DSK files to a real CoCo disk.



Learn some secrets to quick programs on the 68K series!

POSTMASTER:
If undeliverable return to:
FARNA Systems PB
Box 321
WR, GA 31099

If your address is incorrect, send me a postcard!

The Editor's Page

I hope I got your attention with the slight cover changes and the "CoCo 2000" headline. Basically, I am committing to producing this magazine through the year 2000. That is about three more years. Where we go beyond that is up to you, the readers and subscribers. As long as I have interesting things to print and enough subscribers to make the work worthwhile, we'll continue printing even after 2000.

What is worthwhile? We start our fifth year of publication with this issue. In this time, I have posted profits twice, the best year (nothing broke and had to be replaced!) was the last one at about $1,000. Previously I've posted maybe $250 in profits.

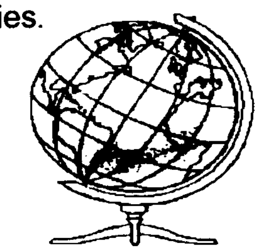
I said "profit", but I mean that very loosely! That is basically what I got out of printing the magazine. I have upgraded my computer equipment several times in order to make publishing easier, and have had to replace a few items that just quit. Of course I use my equipment for other things, so I get a little pay-back by having a little extra money to buy better equipment than I may otherwise have. But I pay myself nothing.

That profit mentioned is basically the magazine's surplus, which is held just in case something does break or I feel that something needs to be upgraded. Very rarely I will make a personal purchase from my business account that I can't write off as a business expense. That is all I get for producing the magazine. And this is mostly offset for the expenses I have at home (like the extra power, some telephone time, etc.).

The first two years had eight issues, the last two six. So that is 28 issues altogether. It takes an average of 30 hours work to produce, assemble, and mail each issue. That comes to a total of 840 hours of labor. Divide $1250 by 840 and you get $1.50 per hour.

While this is hardly enough to think about making the magazine, there are peripheral rewards. One already mentioned is that my computer equipment is a total business write-off. So I get my computer stuff basically for free. Then there are the trips to Chicago and other fests. 90% of the expenses come from my business account. And there are the contacts with subscribers, and the joy of providing something useful for others.

I have to admit, I don't print this magazine for the money! It does a little better than break even. That is enough for what is basically a hobby business. I just hope you appreciate the fact that I am willing to go to so much trouble for so little by continuing to subscribe and support others who work for as little or even less (such as Glenside CoCo Club and Ron Bull, who put on 'fests this year) to support the CoCo and OS-9 hobbyist communities.



the world of 68' micros

Publisher:
FARNA Systems PB
P.O. Box 321
Warner Robins, GA 31099-0321

Editor:
Francis (Frank) G. Swygert

Subscriptions:
US/Mexico: $24 per year
Canada: $30 per year
Overseas: $50 per year (airmail)
Back and single issues are cover price. Overseas add $3.00 one issue, $5.00 two or more for airmail delivery.

The publisher is available via e-mail
dsrtfox@delphi.com

Advertising Rates:
Contact publisher. We have scales to suit every type of business. Special rates for entrepreneurs and "cottage" businesses.

Contributions:
All contributions welcome. Submission constitutes warranty on part of the author that the work is original unless otherwise specified. Publisher reserves the right to edit or reject material without explanation. Editing will be limited to corrections and fitting available space. Authors retain copyright. Submission gives publisher first publication rights and right to reprint in any form with credit given author.

General Information:
Current publication frequency is bimonthly. Frequency and prices subject to change without notice. All opinions expressed herein are those of the individual authors, not necessarily of the publisher. No warranty as to the suitability or operation of any software or hardware modifications is given nor implied under any circumstances. Use of any information in this publication is entirely at the discretion and responsibility of the reader.

Reader's Write...

From a longtime supporter...

Enclosed is an American Money Order to continue my subscription to "the world of 68' micros" magazine. I hope I am not to late.

I have had quite an unusual year (since last Sept. at least!). My husband's health is not to good: angina from heart attacks and other "interior problems". For myself at 81, I have had my share of problems. So we decided to move to a retirement home. It's a wonderful pampered life, but very quiet and satisfying in an independent apartment.

I have to say that we had to exchange my beloved CoCo 2 and 3 plus my wonderful gem of a PC2 for a Compaq with Windows 95. I am not quite used to it (from Disk BASIC and OS-9 to MS-DOS). I find it boring to "click" on all those dialog boxes and lines. CoCo 3 was so fast and I could "program" what I wanted! My husband kept his laptop (Tandy HD1100), he uses it for Deskmate (fast and easy to operate).

So here we are. I have given ALL my Tandy programs, literature, Rainbow, etc., to a specialist in computers who loves the CoCo. He looks after the Compaq and will connect it to the Internet sometime later, after I am more installed in our new apartment.

I shall miss the CoCo to the end of my life, just like I missed the Rainbow when it disappeared.

Frank, all my best wishes go to all of you who have kept faith with the CoCo. As you say, though, space on the desk may push it out...

Thanks for the wonderful work you do, and I hope to be around to enjoy it many, many years to come!

Mrs. Lyone Boult
420 Mackay Street, Apt 405
Ottawa, Ontario K1M 2C4
CANADA

Mrs. Boult, I thank you very much for all the compliments. You have been a patron of the CoCo community for a long time... I know you have every issue I've ever put out and bought nearly every program in my catalog! It is great to learn so much more about you in this letter! My paternal grandmother, whom I love dearly (I'm her favorite also!), turned 82 this year. Your handwriting reminds me very much of hers!

Don't fret about your CoCo though... you can STILL enjoy it on your Compaq! You should have received a copy of the CoCo 3 emulator that I sent you. I discovered that I owed you from the defunct "microdisk" subscription. I hope you find the emulator to be of value! When you are having trouble "teaching" your Compaq to work as you want it to, and not the other way around, fire up the CoCo 3 emulator and let it do all the work! Depending on the speed of your Compaq, you should find the emulator to be faster than the original CoCo 3. This will let you enjoy your CoCo for many more years to come!

I encourage all to send there best wishes to you in your new home, and prayers for both of you for continued good health and speedy recovery from the surgeries you have had.

Year 2000 Revisited

I was glad to see the article by Robert Gault in the latest issue, on the "year 2000" problem. I've been wondering why there's been nothing on that 'til now in the OS-9 community, when it's such a big deal elsewhere.

On that subject, I would like some better programmers to tell me why the following idea wouldn't work:

OK., so there's only one byte available for storing the year in disk identification sectors, file descriptor sectors, and OS-9's direct page. But one byte can count up to 255, which is a lot more years than my CoCo is going to last. Instead of storing in that byte

VAL (RIGHT$ (YEARSTRING$,2))
why not store the value
(VAL(YEARSTRING$))-1900

It's backwardly compatible, it will last until the year 2155, it preserves the integrity of date comparisons between different files. Main question: Will OS-9 choke on a byte greater than 99? Obviously the setime and date modules, and other utilities that print out dates like dir and free will have to be altered, but format and the file creation/modification modules might just work without change, since they probably just copy the year byte from direct page. There's a good chance that a lot of application software won't notice; after all, if a programmer thought the year 2000 would never come, why would he error check for values greater than 99?

Sincerely,
Richard S. Bair
335 Jefferson Ave.
Glencoe, IL 60022-1823

Well Richard, you have me stumped! Looks like it should work, at least until 2155, as you stated! Anything from you OS-9 programmers? Please let us know and it will be printed here!

Need some drives

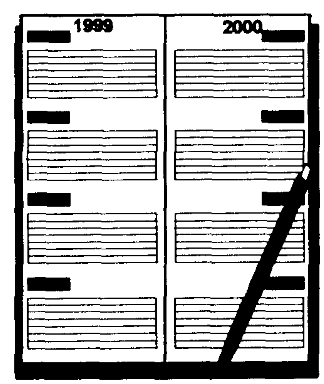
First of all, I hope everything is fine over there. Although I don't use my CoCo3 as much as I used to, I still use it to do all of my data communications chores, to play some games, and to try to learn OS-9, which I think is the best of the multi-tasking operating systems. Hope no one in Microsoft reads this, because I am going to a job interview installing Internet servers running NT 4.0 at the local offices of Microsoft.

I hope that you continue to publish articles regarding hardware projects for the CoCo, and articles about the AT306 systems. I would like to buy one of these when my budget let's me. I find these kinds of articles really interesting.

I am looking for two good 5 1/4 drives, and one 3 1/2 720K drive. I need them to replace the one in my FD-502 nit which has a damaged head, and the 720K 3 1/2 unit to use as a second drive for OS-9. The other 5 1/4 is for an old Leading Edge XT computer that I want to repair. I hope you or any of your readers can help me. Thanks for your help and continue the good job.

Your friend,
Luis E. Tanon
Los Arcos de Suchville, Apt 217
Torre Sur, Peurto Rico 00966

Luis, one source for good used, guaranteed 360K 5.25" and 720K 3.5" drives is Alltronics, 2300 Zanker Road, San Jose, CA 95131-1114, 408-943-9773 (www.alltronics.com). Also try Alltech Electronics, 619-724-2404 (CA, www.allelec.com). Prices range from $5-$25. Make sure you get a 720K 3.5" drive and NOT a 1.4M! The 1.4M may work correctly as a 720K in a PC, but NOT in the CoCo!!



A Missed CoCoFest?

David Baker & Ted Willi

The "First Annual 'Last' CoCoFest of the Clarke County CoCo Club"

Athens, GA (June 23, 1997) – It took a good part of the longest day of this year to hold the First Annual "Last" CoCoFest of the Clarke County Color Computer Club (CCCCC) in Athens on June 21, 1997. When it was all over both club members unanimously declared it had been productive, profitable, and probably worth doing again someday.

Member Ted Willi showed up promptly at 6 PM at the residence of member David Baker, who was already there. Ted brought three CoCo2's, four monochrome monitors, including a Magnavox of exceptionally fine quality, and assorted software, hardware and reading material for consideration. Member Baker tried unsuccessfully to unload a disabled Panasonic 24-pin DMP printer on member Willi. However, it was determined later that Ted had a probable future use for a VGA adapter card that Dave no longer needed, as one is required to get a CoCo3 emulator running on a VGA monitor somewhere pretty soon. Also, David gave Ted a compact color TV, which doubles as a CoCo3 monitor, with the caveat that any attempt to watch any TV show other than Braves games and other culturally uplifting programs will cause any CoCo in the vicinity to crash promptly and permanently.

The highlight of the CCCCC's First Annual "Last" CoCoFest was the swift and sure manner in which one of the club's members (Ted Willi) installed a Baker's CoCo3 emulator on a PC– so competently that it worked correctly right away. Unfortunately, member David, who had been cutting grass and doing other yard work for much of the day, was taking a shower at the time and missed the whole installation. Later, though, both club-members agreed that Jeff Vavasour has done not only a great service for the entire Color Computer community but is also one heckuva smart guy.

Around 9:30 PM the members took a break for refreshments, which consisted of some sloppy hotdogs with Vidalia onions and chips and some fat-free but not calorie-free ice cream, and viewed the last few innings of a somewhat sloppily played Braves-Phillies game.

After the lunch break the entire membership returned to the Computer Room, scene of the hottest activity at the CoCoFest, where David beamed aboard the Internet and showed Ted the mother of all CoCo web sites from Saskatoon, Saskatchewan and pointed out the link to Al Dages's web page. A side excursion was made to member Baker's Usenet subscriptions, where he and willi read several hundred postings from comp.os.os9, comp.sys.m6809, bit.listserv.coco, and comp.sys.tandy. Afterwards Ted logged onto his Delphi account by using the Telnet facility on Dave's PC.

Finally after midnight the club members gathered up their CoCo gear to head home (except for anyone who was already there) and heartily agreed that this had been the very best First Annual "Last" CCCCC CoCoFest ever held in Athens! And so, at last the Fest wound down with that unique CoCo technolust assuaged for the moment. But each club member knew in his heart that, like a stuck disk drive, the compulsion to meet and swap CoCo stuff is one that never truly times out!

Note: The Clarke County Color Computer Club holds its meetings whenever its members can get around to it. One of its most active members will be moving to Atlanta soon, which proves that you don't have to live in Athens to be a member. In fact, if you already belong to any CoCo club anywhere, you are automatically eligible for membership in the CCCCC, which has already grown to two members in only the last few years, probably because there are no annual dues. This article has been presented in lieu of the long-planned club newsletter to be published someday maybe. (editor: Both members are subscribers, by the way!)

From: *Dennis Bathory-Kitsz*

Hi folks! I've been hiding out in Vermont, but since it's the 10th anniversary of my company Green Mountain Micro's demise, I thought it might be time to put in an appearance here.

About 150 copies of 'Learning the 6809' (book only) remain, which I'd be happy to offer at $10 postpaid to anyone interested. If at least 10 people also want the original tapes, I'd be pleased to make up a set of those as well.

One of these days I'll tell my own tale ... amusing indeed...

Dennis Bathory-Kitsz
RD 2 Box 2770
Cox Brook Road
Northfield, Vermont 05663

<bathory@maltedmedia.com>
Malted/Media:
http://www.maltedmedia.com/

operating system nine

Rick Ulland

A paper on the CoCo4 - RFC

Note: This was originally intended for the Pennsylvania CoCoFest hosted by Ron Bull. Unfortunately, I never got there! Reprinted as originally typed.

Why?

I guess we need a purpose statement. The theory is, modem computers contain CPU's that far surpass their users needs. But limited to One True Serial Processor, these machines often leave their user staring at a hourglass or piling work to the side until the 'right program' is loaded. Even with it in que, the amount of work that can be paralleled is small.

This situation isn't likely to change as long as the only people capable of experimenting with the technology are those needing to protect next quarters net, but a modern CPU is just too fast and compact to play with. With the CoCo, we have hardware that's old enough, slow enough, and 9/10ths of the needed opsys available at close to free.

At the time of the last Chicago CoCoFest, we had quite a talk on the CoCo4 project. Lots of good ideas where tossed around, but response on the mail list has been.... less than astounding. I think we needed a 2nd hour just to organize the project!

The report from Milwaukee-

Project B (I think it was B) was a GIME emulator running on a larger CPU. Some specs tossed about at the fest- we'd probably need a CPU-32 core to avoid the nastiness of even word boundaries, and speed/sync would be a problem- nothing too earth shattering there. Carl has done some additional work since the fest, but he's got the IDE project to sweat over. Not being familiar with 68K code or the chips true grit, I'm eagerly awaiting news from PennFest. Feel free to step up.

Project A was a co-CPU I/O board, designed to unload the motherboard itself from the cycles needed to emulate a keyboard with PIA, mouse interface with software timer, and the like. Another worthwhile project but again, no feedback. I think we've got the logical conclusion of that idea below... and one that can make good use of existing designs. I'm hoping if anyone builds a true 'PA' they'll let us steal the details! But before that, something I've got data on, Project C.

IRQ Controller

My first hardware project has been the IRQ controller. To recap, the idea is to have each IRQ driven device provide two datum- the normal CART IRQ as an alert (also acting to compatible existing IRQ-poll software) and an additional discrete byte that describes the source of the interrupt. It should be relatively simple to cobble OS-9 to read the hardware provided ident byte rather than go through it's polling routine- a substantial speed increase.

The fine points- first, everything except the clock would leave a telltale byte in the IRQ register. In other words, if a CART is received and the register is 0, you've just gotten a clock IRQ. Second, the register is cleared on reading, so we could probably cheat a little bit- instead of ignoring IRQ during an IRQ service, check for nonzero IRQ buffer before returning to normal processing. This would catch one extra of anything but clock, and cost 0 cycles in mid-routine. We could possibly FIFO the IRQ register, and adopt a better late than never IRQ scheme. In any case, this device should be installable and useful on a CoCo3. It will be an absolute requirement on the CoCo4, due to the raw number of IRQ that will be generated.

Moving 4 ward.

My next idea is to just start something and see who joins in. The following is a general theme for a true 'CoCo4' that could amount to more that stirring through the ashes for another spark. Consider this a request for comments. What I'd like to do is solicit everyone's ideas, publish another RFC, etc., until we end up with a real specification. Not a wish list- we know VGA graphics and 3 Gig IDE drives are nice. What we need is a service manual. Soft and hard ware problems can be studied in reference to a known spec, which will be changed to solve the problems... once everybody's happy (famous last words) we'll build it. I've got way too many potential specs to list here, so expect a quick gloss over the top and a big follow-up article.

MultiCoCo

Once we have a usable CPU/CPU buss the rest will come. I propose a box that accepts single CPU 6x09 cards. The builder writes the code to make said 6x09 do anything, either as OS-9 code or stand-alone assembler, adds the needed hardware and plugs it into the box, which helpfully provides OS-9 system services (scheduling, file handling, etc.). This would combine the ease of programming and attaching devices to a CoCo2 with the management skills of OS-9.

Design for the main board (sysCPU) has been purposely left for last. In the early stages, we can perhaps get away with a hacked CoCo as master CPU, but of course this will change as we determine what's needed. Anticipate a 16 bit (RAM buss sized) CPU to CPU buss.

A generic primary expansion unit (appCPU) might be a single 6x09, PIA, and 2 64 K banks of dram. Within the CPU's 128K, the ram is remappable ('remotely' by syscall to sysCPU) though of course each complete 128K segment appears immobile to OS-9. The 'bank switch' bit is read/writable by both CPUs and appCPU has 4 basic states-

1) running map A code
2) running map B code
3) HALTed pending system access (Both task halted, read DAT for pending request)
4) both maps busy

The interCPU buss will be tristated for any bank the appCPU is using. OS-9 could override this by halting the remote CPU and forcing state 3 (danger to app).

Such a card has two uses- it can drive a bit of hardware via PIA, or it can take some code load off of the main CPU. The OS-9 code loaded into this board could be:

mapA:
1) shelldrv
2) shell or sh09
3) application modules.
4) Data (Could also use mapB- see sh09)

mapB:
1) grfdrv/vgadrv
2) screen

1) more OS-9/PIAdrv
2) OS-9 data

1) 6x09 assy (non OS-9)
2) data swap to OS-9

The idea is you have one 64K process space, and one 64K map to do with as you will- free of the normal scheduling constraints imposed by OS-9 (This scheme begs for a 512K version). An OS-9 task module controls the timing and signals for the 'alien' task map.

OS-9 on the daughterboard

Shelldrv's main task is to pretend to be OS-9 to the application process. It needs to intercept anything a normal shell would send the system and halt the app CPU after pushing the CPU state- once restarted, it has to poke the app CPU into whatever state OS-9 returned (ram changes already done via dual port) and continue. Under this, one could use shell to run normal OS-9 code, using the main opsys i/o and scheduling.

To implement non-OS-9 pages, we'd need a shell replacement capable of loading a code block into the alternate task map (or picking up the ROM) and riding head on it from the OS-9 side. In this document, it's called sh09. Sh09 would also need to manage (OS-9ify) the immobile data blocks used in the alternate map.

Basically, any system access to mapB would be under sh09's control leaving it outside the normal flow of OS-9. Sh09 would have to define the percentage of time it's particular appCPU spent running the foreign code, catch local IRQ before they get to shelldrv and OS-9- and pretend it's sleeping part of the time.

The foreign code has a couple of ways to interface. First, it can simply write to the data area and let sh09 find it (and the reverse). It could also use the local IRQ system to run the (OS-9/sh09) IRQ handler for time critical stuff. We'll need some heavy systems programming dudes to ponder on the OS-9/sh09/assembly details.

OS-9 on the motherboard

The task switching routine has to be changed. Until it runs out of appCPU's, OS-9 won't be spending any CPU time running mainline process code, instead there is going to be a wash of syscall requests. The scheduler will have to go by time in syscall rather than ticks per timeslice. Just doing it will be a task- when a syscall or other IRQ comes in, the system has to save itself and the entire sysCPU state, load the whole appCPU state, page in the remote RAM, finally run the code and get back. This needs to be a terribly efficient bit of code, much of the system's time will be spent here. Alan (Dekok... of Nitro fame)?

The ram DAT has to understand it's pool and the remote, semi-fixed address blocks of the app CPUs. Sh09 can handle the non-OS-9 code, but the OS-9 side is itself sort of immobile. How to tell OS-9 a process sometimes has to load in the range aaa0000-aaaFFFF? (Also, it's data is always at a static location). There also has to be a way to page local and remote RAM around quickly so process can be swapped among the various CPU's RAM areas. Even disk data would be coming in this way! Pipes need to be beefed up to use the fast swapper for interCPU pipes.

And if that's not enough, redirection has to be expanded to include CPUs as well as devices- and we need the companion Xprocs to identify which CPU a task is on.

Possible Initial Card Designs:

Smart DMA type hard/floppy disk controller
Same thing in a SCSI controller
Smart VGA card (also emulate GIME/grfdrv res)
Multiport card for 'other i/o'- stack the keyboard controllers and printer ports on one CPU
'Mega ports' like a 16550 with built in PPP/slip
Rack o' CPU (perhaps a half dozen 6x09's with RAM, no I/O)
Multiple Math co-proc (perhaps in the rack o' with the MRola math ROM)

Main System CPU-

There is a major choice here- perhaps 2 competing systems. The first option is to capture 'current' (as in, not forgotten yet) Lvl2 knowledge in a 6x09 system controller. If the video is moved to an app, one of the traditional 6x09 problems (small, slow gfx) might be solved... with a 6x09!

The second option is to use a 68K. Lots more ommph at the base and we could use a VGA card. The problem being OS-9 68K is harder to come by and hasn't been hacked as much.

So the two sysCPU sections are:

6809: There's no reason why a CoCo couldn't serve as the development engine- in fact, there are reasons to do so. During testing, a few known working I/O systems can be handy, and bits of the motherboard can be written out of the opsys as they become obsolete. As a product, the add to CoCo concept means a system can be bought one card at a time. The user could stack 12 CPUs on the CoCos built in I/O, or let it's single CPU run free thanks to the smart drive controller and i/o board. Or anyplace between. Eventually, a 6x09 motherboard of CPU, ram, monster DAT, and not much else could be developed.

68xxx: OSK emulating OS-9? Or 6x09s passing OSK style calls? If shell and shldrv are rewritten to 'look' OSK, we might as well go right to the multi-PPC engine and look for investors;-) If they appear OS9/6809, CoCo programs would be compatible....

More later!

Rick Ulland
CoNect and 'operating system 9'
1629 South 61st Street
West Allis WI 53214
pulland@omnifest.usm.edu

CoCo4x hardware coordinator

Zen of Color Computer Programming

Chet Simpson

Interrupts and sound

A Little Recap.

If you thought something was missing from the last issue, you just might be right. Yes, I missed the deadline. During the time I needed to get the article to Frank, I was busy relocating from Des Moines, Iowa (where I worked for Microware in the MPEG group) all the way back to the West coast to sunny California! I'm now totally moved and have settled down into my job (which is still in the Interactive TV arena). I'm finding myself more and more attracted to this state (even though I seem to have an allergic reaction to earth quakes, but doesn't everybody?), especially since I haven't seen a SINGLE drop of rain since I have been here.

Remember, in the first article I mentioned a challenge that was issued by a friend of mine. That challenge was to create [for the CoCo III] a duplicate of the MM/1 game "Gold Runner 2000." Well, unfortunately, I lost the challenge, but by technicality only. Seems that my move coincided with the deadline of having the game "complete." But never mind that, you can see the results of the game in it's final release as Digger II: Return of the Saint (that's a product plug by the way!).

What about the good stuff.

In the last article, I mention playing real-time digital music like those used by nifty music programs running under MS-DOS. By real time music, I mean that digitized samples of musical instruments are played back with the various notes and sound effects computed while the samples are being played back.. Almost everyone that I have talked to about this said that it was impossible to do. Until John Kowalski released his CoCo III .MOD player. Unfortunately, doing all of those calculations to get the correct musical notes is very taxing on the CPU and leaves little time to do anything else. Because of this, I started looking at alternative ways to accomplish the same effect but without using a tremendous amount of CPU time. The following starts off a 3 part series on how to get up to 4 voices real time digital music and still have at least HALF of the CPU left over.

A little background.

MODule files originated on the Commodore Amiga. It allowed the Amiga to play back digital music in 4 distinct voices without any special sound hardware other than a Digital to Analog Converter (DAC).

The original MOD files were made up of several parts; a simple header, instrument samples (up to 16), "pattern" data and a list of what order to play the "patterns" at (allowing patterns to be reused). The most important part of a MOD file is the pattern data as it described which sound sample and note to play it at for each voice. A decoded MOD file might look something like this:

Sample 1: Snare drum
Sample 2: Bass Drum
Sample 3: Flute
Sample 4: Symbol

Pattern Data:
(Displayed as sample:note/octave for all voices)

```
     Voice 1 | Voice 2 <
  -----------+----------->
  1] 1 : C3  | 3 : D1 <
  2] ......  | .... >
  3] 2 : C3  | 3 : A4 <
  4] ......  | .... >
  5] 1 : C3  | 3 : D4 <
  6] ......  | .... >
  7] 4 : C3  | 3 : E3 <
  8] ......  | .... |
```

For now, lets look at voice 1. The first note in the pattern plays sample number 1 (the snare drum) as note C in octave 3. If you look at the notes in most MOD players, you will find that this is a very common occurrence, especially with rhythm (drum) tracks. This is because most samples are recorded as a C note (no not a $100 bill) in octave 3, also known as Middle-C. This allows the samples to be "transposed" or changed into other notes and octaves in a known manner to the composer. Keep in mind that it is not a requirement that the samples be recorded in Middle-C. You music buffs out there may have already noticed that there is no specification for the duration of the note. Line 2 signifies that the note does not change and therefore continues to play until the end of the sample. If the end of the sample is reached, nothing is played until a new sample is specified.

But what about the CoCo you ask.

During the final development stages of Digger II, I wanted to add something really special to it using the little but of memory that I had left over. I started to look at MOD type files (mainly because I have several megabytes of samples), but wondered if it would be done in a fast efficient way. So I wrote a small program that really ripped into the MOD file, retrieved the size of each sample used along with how many notes and octaves it was played at. I was quite amazed at how little some of the small to medium sized MODS utilized the capabilities of the MOD players. While some of the smaller ones only used 100k, quite a few of the medium sized ones only needed 250k to 300k of total memory. This of course does not take into account many of the special effects (such as volume changes and Vibrato) that a MOD can do. Needless to say, this got me very excited! My (then original) goal was to push out 2 voice real-time digital music but ended up being 4 (but 6 is possible).

Using the Timer FIRQ to play sound at 8 kilohertz.

The best part about playing back sound on the CoCo III is that we can use the timer interrupt of the GIME chip to give us a consistent playback rate. You see, the GIME chip allows you to specify that an interrupt will occur within a specific amount of time. This can be done on either the IRQ or the FIRQ (Fast IRQ). Since we don't want to use a whole lot of CPU time, we will use the FIRQ, because unlike the normal IRQ, the FIRQ does not save the state of all registers which makes it FAST. What makes the GIME even nicer is that it automatically restarts the timer after the interrupt has been acknowledged.

In order to make the GIME chip send us an interrupt, all we need to do the following (See Listing 1):

1. First we want to make sure that the FIRQ vector points to our FIRQ handler routine. If an FIRQ happens and the vector points to nowhere, the machine might crash.

2. Normally, both the IRQ and FIRQ are generated the same way they are in the CoCo II in order to keep compatibility with older software. So the first thing we need to do is tell the GIME we want to use its FIRQ mode. We do this by setting the FIRQ enable bit which is bit 4 ($10) of the GIME INIT 0 register ($ff90).

3. Next, tell the GIME which event to trigger an FIRQ on [The GIME supports triggering an FIRQ for the keyboard, serial port and others hardware]. Since we want a timer interrupt, this is done by setting bit 5 ($10) at of the GIME FIRQ Enable register ($ff93).

4. Before we do anything else, we need to make sure that we receive an FIRQ at the correct time. So we first select the timing method (70ns) by setting bit 5 ($20) of the GIMEs INIT1 register ($ff91) and then by setting the time interval to trigger an FIRQ. Remember that the timer automatically restarts when the MSB (Most significant byte) of the timer value is set.

5. The final part of the initialization that needs to be done is to trigger the GIME to start sending us interrupts. To do this, we simply read the value of the GIMEs FIRQ enable register ($ff93). Of course, we can also do any number of things between those steps, which our demo code does in order

to turn the sound on.

All that is left is to actually PLAY the sound. Since our interrupt routine gets called around 8000 times per second, we want it to be as fast as possible. This is done with a little optimization and the use of self modifying code. Don't worry, it only changes a couple of address and data bytes, none of the code actually changes. Since this is a very short routine, we'll step through it (It is also include at the end of the article as Listing 2).

How the sound is played.

The sound routine is very simple. It starts playing at address $8000. Every time it plays 256 bytes (1 page), it decrements a counter to help it keep track of where in the buffer it is. Once it has played 32 pages (8k), it starts playing at location $8000 again.

Since the FIRQ hardware of the 6809 does not save the states of the registers (with the exception of CC and PC) we need to save the registers that we do use. Thankfully we only use register A. This is stored into the routine itself so that we can do an immediate load of the value back into A. This is much faster than doing a PSHS/PULS of the register.

```
sirqrt
    sta smc+1    * save ACCA
```

The next thing we do is get a byte from our sample buffer and send it out the sound port.

```
soffst
    lda $8000  * get sound byte
    sta PIA1   * send sound out
```

Now we get back to the self modifying code which is a little bit tricky. When we get the byte from the sound buffer, we don't use any indirect addressing. If we did, we'd be wasting some CPU cycles. To get around this, we use extended addressing. But since we have to increment that address. First we increment the LSB of the address. When the LSB reaches 255, the next time it is incremented it goes back to zero. This is handy in that both a flag has been set in CC register and we have just played .

```
inc soffst+2
    * increment LSB of offset
    bne endit
    * no overflow. Reset FIRQ
      and return
```

Since we have just played exactly 1 page, we now do 2 things, the first is increment the MSB of the address. This keeps us from playing the same 256 bytes all of the time. The next thing we do is decrement our page counter. If this reaches 0, we know we are at the end of the sample buffer and we need to reset both the page counter (back to 32) and the address (back to $8000). When we reset the sample address back to $8000, you'll notice that only the MSB of the address is set ($80), this is because the LSB is already at $0 (convenient huh).

```
inc soffst+1
   * increment MSB of offset
bne endit   * No
dec count
  * Are we done playing 8k block?
bne endit    * No
lda #$20    * Get new count value
sta count    * set it
lda #$80
   * Get MSB of sample address
sta soffst+1  * set it
```

Next we need to reset the interrupt flag and do other cleanup. To reset the interrupt, all we need to do it read the value of the GIMEs FIRQ enable register. We could use TST but it is 1 cycle longer than LDA. Next we restore register A back to its original value. Remember the first line of the FIRQ handler stored it here. The we do an RTI (Return from Interrupt). Do NOT use RTS!

```
endit
lda FIRQENR  * reset FIRQ status
smc
lda #$ff     * Restore ACCA
rti          * return
```

Caveats.

Keep in mind that this is not a complete set of routines for playing back sound. It only plays a sound sample from 1 area of memory. When using these routines make sure that you have the IRQ disabled (keep the FIRQ enabled or it won't work). This will keep the IRQ routine in DECB from crashing the machine.

Contact information.

Yes, I have moved! Fan mail, chocolates (packed in dry ice please), mail bombs, etc. can now be sent to:

Chet Simpson
5525 Canoga Ave. #320
Woodland Hills, Ca 91367
Or if you prefer SPAM:
medialink@delphi.com

Next time.

In the next issue we will combine both the IRQ and the FIRQ for playing back more than one sound and we'll get started on what we need in order to play real-time digital music on our "slow" 2mhz CoCo.

Listing 1 (Initialize GIME and Sound):

```
PIA0 equ $ff00
DAPORT   equ $ff20
PIA1 equ $ff20
INIT 0    equ $ff90  * Initialization register 0
INIT1    equ $ff91   * Initialization register 1
IRQEN    equ $ff92   * Interrupt request
enable register (IRQ)
FIRQEN    equ $ff93   * FIRQ enable register
TIMSB    equ $ff94   * Timer MSB
TILSB    equ $ff95   * Timer LSB

FEN equ $10      * GIME FIRQ enable
TINS equ $20     * Timer input
TMR  equ  $20     * Timer FIRQ enable

   pshs cc,x,d   * save cc registers
   orcc #$50    * Disable interrupts
**********************************
* Set up hardware FIRQ vector
**********************************
   lda #$7e     * use JMP operande
   ldx #firqrt  * point to FIRQ routine
   sta FRQVEC    * set the FIRQ vector
   stx FRQVEC+1  * Set FIRQ handler
**********************************
* Set up GIME to use the FIRQ
**********************************
   lda #$4c+FEN  * Enable GIME FIRQ
   sta INIT 0    * set it
   lda #TINS     * select 70ns
   sta INIT1     * set init register
   lda #TMR      * select timer irq
   sta FIRQEN    * use FIRQ
   ldd #470      * Get timer (8k) count down
           * value
   stb TILSB     * set timer lsb
   sta TIMSB     * set timer msb and start
           * counter
**********************************
* Reset values for FIRQ routine
**********************************
   lda #$20     * Play 32 256 byte pages
   sta count    * set it
   ldd #$8000   * Start playing at $8000
   std soffst+1 * set it
**********************************
* Turn on 6 bit sound
**********************************
   lda PIA0+1    * select sound out
   anda #$f7     * reset MUX bit
   sta PIA0+1    * set it
   lda PIA0+3    * select sound out
   anda #$f7     * reset MUX bit
   sta PIA0+3    * set it
   lda PIA1+3    * Get PIA
   ora #$08      * select 6bit sound
   sta PIA1+3    * set it
   clr $ffa4     * Reset MMU to play from
   lda FIRQEN    * reset FIRQ
   puls d,x,cc,pc * return
```

Listing 2 (Handle FIRQ):

```
count    fcb $00
firqrt   sta smc+1 * save ACCA
soffst   lda $8000 * get sound byte
   sta PIA1      * send sound out
   inc soffst+2  * increment LSB of offset
   bne endit     * No overflow
   inc soffst+1  * increment MSB of offset
   dec count     * Are we done playing block?
   bne endit     * No
   inc $ffa4     * Increment
   lda #$20      * Get new count value;
   sta count     * set it
endit lda FIRQENR * reset FIRQ status
smc lda #$ff     * Restore ACCA
   rti           * return
```

Transfer .DSK Files!

Bob Devries

Use CoCo emulator .DSK files on your CoCo or the emulator!

RSDSK and OS9DSK are two programs I wrote recently in response to messages on the COCO list on PRINCETON (coco-list@princeton.edu). There had been talk of the disk image files (.DSK) that are created for Jeff Vavasour's COCO emulator. These files are an exact image of a Color Computer disk, track-by-track, sector-by-sector. For the Disk Extended Basic image files, they are 161280 bytes long, for 35 track, 18 sectors per track disk images. The disk images used under OS-9 with the emulator may, of course, be any size.

These files are now being uploaded to COCO FTP sites like OS9ARCHIVE.RTSI.COM. Problem is, they are only useful to people who have an IBM PC or clone. People who only have a COCO and/or one of the OS-9/68000 based computers such as the MM/1, could not get at the information stored in those files. Imagine having a computer disk, but not having the OS that it was created on!

So, being the sort of person I am, I decided to see if I could write a program to 'read' the image files under OS-9. I started off programming it on my MM/1, and when I had it working to my satisfaction there, I did the necessary modifications to make it compile under BOTH OS-9/6809 and OS-9/68000.

I started with the DECB image files, which I thought would be the easiest, since there's only a single directory, with a maximum length of 16 sectors (yes, I know it should be 9, but I wanted to allow for disks larger than the standard 35 tracks), 128 entries maximum. Also, I had many times written BASIC programs to read the directory from a DECB disk.

It turned out to be easier to rewrite from scratch, than to 'copy' the BASIC program. Of course, it was to be written in C, which is what I write in most.

About a day and a half later, I had a program which would do a directory of the disk, and another day later I had the program completed, with capability to do a directory, copy a single file FROM the image file, and make a procedure (script) to get all the files from the image file. I didn't then, and still have no intention to write one to copy TO the image file.

Usage of the RSDSK program:

RSDSK -dir filename.DSK

or

RSDSK -get filename.DSK DECB.filename OS9.filename

or

RSDSK -proc filename.DSK

The first example will produce a directory of the emulator image file 'filename.DSK'. Note the full filename is necessary, including the '.DSK'. The second example will copy the file called DECB.filename from the emulator image file 'filename.DSK' to the OS-9 pathname 'OS9.filename'. The third example is something I thought should have been done by the author of the program 'RSDOS', which reads COCO DECB disks under OS-9. It produces a procedure file which will copy all the files from the emulator image file 'filename.DSK', much like the 'dsave' program does under OS-9. This procedure can then be edited, or executed as a script file, or piped to shell from the same command line.

Having done all that for DECB emulator image files, I was left with nothing to read OS-9 emulator files. So that was the next challenge.

It was both much easier, and much harder. Easier because the directory structures and other code is already in place to use in the C compiler. Harder because OS-9 has a tree-like directory structure. To transverse down the directory tree required 'recursion', which was entirely new to me.

Anyway, a few days later, I had a working program, and promptly uploaded it to the FTP site, only to be tripped up by Jeff Vavasour himself, with an example disk image file that was bundled with the new version of the CoCo3 emulator. It had a set of conditions that I had not allowed for, and my program promptly went haywire. It took me long hours to find the problem.

The syntax of the OS9DSK program is much the same as the RSDSK one, with the addition of a 'directory' argument.

OS9DSK -dir filename.DSK dirpathname

or

OS9DSK -get filename.DSK imagepathname os9pathname

or

OS9DSK -proc filename.DSK dirpathname

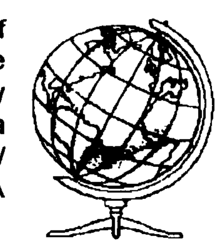
In the examples, 'dirpathname' is the optional pathname to the directory required (like CMDS or USR/VED/CMDS). In example one, the output is much like the dir e (or dir -e) command. In example two, the 'imagepathname' is the pathname (as shown by the '-dir' command) of the wanted file in the emulator disk image file, and os9pathname is the place where you want the file copied to. In example three, a procedure file is created (again like the 'dsave' command). A starting directory can optionally be used. All subdirectories encountered are descended into. This output can be edited, run as a shell script, or piped to shell.

Where do you get it?

Currently the files are on the OS9ARCHIVE.RTSI.COM FTP site in the directory "/OS9/incoming/coco" and "/OS9/incoming/osk". They are called 'rsdsk.lzh' and 'os9dsk.lzh'. The programs are freely distributable, and source code is provided for you to learn from. It is NOT well commented, but if you need help understanding them, or have any bug reports, I'm available on the Internet at:

bdevries@gil.com.au
(Australia)

NOTE: Copies of these programs are available on disk by sending $5 for a shipping/handling/copy fee to FARNA Systems.



G-Windows Error

Steve Adams

Mixing different versions of G-Windows and G-View

Question: *"We're developing a G-Windows app using G-View on a MVME162 box. The target is a 68360-based system running OS-9 v3.0. On the target we're running the newest version of G-Windows (ed. 62 if I recall correctly), but on the VME162 box we're compiling under G-Windows ed. 54 (once again IIRC.)"*

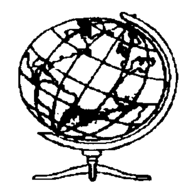
There is one discontinuity in the development of G-Windows. Except for this one problem that I had to work around, every change to G-Windows has been fully backward compatible.

You cannot mix versions of G-Windows and G-View if they lie on opposite sides of the edition #62 boundary.

Now an explanation of the cause of the discontinuity: With V 1.2 of the OS9/68000 Ultra C compiler, Microware changed the variable definitions in the cstart.r file. Variables like 'errno' are defined in cstart.r, and previously were always in the same position in a program's data space. This was important when using sub-routine modules, because it gave them a way to access the 'errno' variable. When I originally designed gadget subroutine modules, I followed example source code from Microware that relied on the variable definitions in cstart.r.

The cstart.r file with OSK Ultra C (V1.2 and later) has the 'errno' and other variables declared in a different order, so the errno variable ended up in a different location in the variable space. When gadgets thought they were writing to the 'errno' variable, they were really writing to a stack checking variable. This made programs quit randomly with a stack overflow error. There were probably other related problems as well.

I couldn't very well ask Microware to take back all copies of their Ultra C compiler and change back to the old format, so I had to change the gadget structure so gadgets would not reference these variables. I did put in a check so when incompatible programs and gadgets are used together, they abort immediately rather than wait for a strange error later on. I apologize for not using a more descriptive error code, but there isn't one.



SOURCES!

I would really like to run this as a regular column. What I am looking for is sources for hard to find and bargain items for CoCo, 68K, and general computer use. If you find a treasure trove of good, inexpensive parts, let me know!

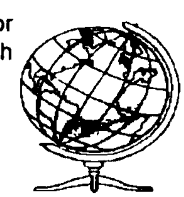
CoCo and MM/1 RGB Monitors!

I think I have found a source for the great Magnavox 8CM515 monitors! These were badged as "Magnavox Professional RGB Monitor 80". They are 14" monitors with Analog RGB (for the CoCo 3, MM/1, and Amiga), digital RGB (for IBM/Clone CGA output) and combined and separate composite (combined for CoCo or VCR, separate for some Commodore models). I've used one of these for years, and they were highly recommended by Marty Goodman and other CoCo gurus. It is possible these are the larger dot pitch Magnavox 8CM505, but at this price ($45!!) even it would be a great deal! All are used with a 90 day warranty. Shipping should run around $15.

Computer Recyclers
972-245-3008
crecycle@airmail.net

If you still have a CoCo and an IBM compatible machine on your desk, here is a good solution.. one monitor for both! These are refurbished NEC multisyncs that sync from 15.75-36KHz... low enough for the CoCo3 yet high enough for 1024x768 VGA graphics! $175 each.

Pikul & Associates
101 Glenfield Drive
Festus, MO 63028
314-937-0335



680x0 Series Performance Guide

Andrei Moutchkine

Tips and tricks for writing fast 68K code.

Once upon a time, I needed to develop a graphics library for an embedded project in 68K assembly. I remember how I couldn't find much hints on how to write fast 68000 code anywhere back then, so I decided to write down some tricks I learned from my mistakes to spare someone else from making them. Most of these are faster/smaller alternatives to 'trivial' implementations of common tasks discovered in "Wait a minute. There is a better way to do this!" fashion.

This guide applies to original 68000-based and CPU32 cores most common to embedded 68K systems. Everything in here could or could not be true for bigger 68K processors and ColdFires. If you find something to add to this guide or any bugs or comments drop me a note at:

muchandr@csua.berkeley.edu

The latest version of this file is available at:

http://www.csua.berkeley.edu/~muchandr/m68k

The target audience is people already intimately familiar with 68000 instruction set. It is not meant as an introduction.

I. FUNCTION CALLS AND CONTROL

1. If you are writing a function that has no local variables, don't create an empty stack frame. I mean skip that 'link An,#0'. Note that your stack will be one word shorter then.

2. bsr is better than jsr. bsr.b/bcc.b is better than bsr.w/bcc.w, which is in turn better than bsr.l/bcc.l. Always try the shortest possible displacement first. This is because an 8-bit offset fits into branch instruction, 16-bit uses an extension word and 32-bit uses two. Any instruction with unqualified size could be silently assembled into .w by default, which sucks for branches. Check your assembler's manual. I don't recommend leaving the size unqualified for any instruction that has more than one possible size.

3. A combination of individual move's (not necessarily all of the same size) can be faster than a movem. Count the clocks. The breaking point is usually around 4-5 on 68331.

4. Nothing prevents you from having several entry points into the same procedure or having several rts' except for scorn of structural programming minions.

5. Ignore the procedure calling conventions inside your own code. Consider saving registers you wish to preserve across a procedure call in unused address registers. If you are working with words on a CPU32, alternating moves to memory with swaps is very fast too because swaps have a large head. Effectively, you will be saving every odd register in memory and every even one in upper word of itself:

```
move.x Dn,memory
swap   Dm
```

II. ADDRESSING

1. 'addq.x #offset, An' is better than 'lea offset(An),An'. In all other cases try to use lea for address calculations over adds. Depending on which processor you have, lea will be at least the same speed as combination of adds and shifts for any address with nonzero offset. Note that lea can be abused to perform a lot of math other than address calculation at once.

2. Any operation on an address register affects the entire register regardless of the size of this operation. If you are finding yourself using something like 'adda.l #constant,An' you are probably wrong and adda.w will do as good of a job. (only faster)

III. INSTRUCTION SET USE

1. 68000 is not a load/store instruction set. Leave the values used only once or twice in memory and access them directly by instruction doing arithmetic on it. (i.e. there is no need to move them into registers first) I imagine this is a common pitfall for people with high-performance modern RISC CPU background. I repeatedly caught myself going through unnecessary trouble to avoid going to memory at any cost. People who mostly did 8-bit stack based assembly before are probable to make the opposite mistake and underutilize 68K's (mostly) orthogonal register set.

2. tst.x is better than 'cmpi.x #0' or, god forbid, 'btst.l #31'/'btst.w #15'/'btst.b #7'.

3. Use moveq instead of move where you can. It is easy to forget that moveq accepts a much larger range of numbers than addq/subq.

4. On a CPU32, try to order your instructions so that you follow instructions with an operand fetch to memory/trailing write immediately by dbcc's (head 6) or shifts/rotates (head 4+) or swaps (head 4) or bit ops (head 2-4 on a register) or any instruction with non-register source (head 3+) or short branches/exchanges (head 2) to maximize instruction overlap and utilize CPU32's mini-pipeline well. Consult the tables in section 8 of CPU32 reference manual for exact timings.

5. You don't get to show off the xor trick. There is an exchange instruction.

6. Don't forget that moves set the condition codes too, not only arithmetics. (i.e. no need to test something you just loaded). Any operations on address registers do not touch CC's however.

7. On a CPU32, use the '68010 loop mode' for bulk transfers. To do so, set up a dbcc loop that branches back to the previous instruction. Any single-word instruction will do, but you probably want 'move.x memory, memory' or 'move.x Rn,memory' type thing. Complicated addressing modes will disable the loop mode, because they require extension word(s). Note that this includes the immediate mode an anything but 'quick' instructions. The loop mode is essentially an instruction cache 3 words (2 instructions) large.

8. There is no reason why dbcc should always loop backward. Consider this piece of pseudocode:

```
while(counter - -)
    if (counter even)
        do foo
    else # odd
        do bar
```

(This kind of logic would be very common on a device that uses a graphics subsystem with 4-bit color depth)

Here you can eliminate constantly checking for a condition inside the loop by having code segments for foo and bar terminate with dbf's branching to the alternate segment like this:

```
foo:
    ...
    dbf Dcounter,bar
bar:
    ...
    dbf Dcounter,foo
```

9. Use add.x Dn,Dn to do a multiply by 2/left shift by one. Use add twice for x4/left shift by 2. It is still faster than shifting this way unless you are on an original 68000-based core and the data is long (.l).

10. Often extend instructions can be used to clear upper bits of a register more efficiently than something like 'andi.x #mask,Dn'. Just make sure that bit 7,15 or 31 (the sign bit) is always zero in the largest possible value you can have.

11. Think of scc as of conditional move of -1. Followed by add/sub you can nicely add/subract 1 conditionally without any branching.

continued on page 17

CoCo3 Extended Memory Secrets Part 3

Herbert Enzman

The FF9x Registers

Now that you have some experience under your belt, it is time to play with some of the $FF9x registers. We can use Z-BUG to change values in these registers and see the results on the screen. I will WARN you now that when changing some of these registers, you will change WHERE and HOW the screen is displayed, and it will APPEAR that Z-BUG has crashed, but it is still in control! You will just be TYPING BLIND, and will have type carefully when the screen goes bonkers.

It took some time to compile this information because most of the $FF9x registers are WRITE ONLY, and when reading them with Z-BUG, only $1B is returned. This made it difficult to find out WHAT value is NORMAL for that particular register. The tables in this tutorial will save you a lot of tedious work.

The information presented here is NOT the last word on the subject, only a start. TABLE 10 for example, only lists SOME information for $FF99 in text mode. Some values for $FF99 will just repeat what is already in the table, and some will display 1 line into 2, because SECB is setup for the 80 column display at this time. There are times when several $FF9x registers have to be changed for certain display modes, as can be seen in TABLE 11.

Why there are two tables for GFX is uncertain. $FF99 graphics modes will not be covered at this time, but will be in a future installment. I won't waste space or time to cover information that has already been presented on the $FF9x registers; The scope of this tutorial is to cover WHAT VALUES to PLUG into them to get some USEFUL work.

I'll start with $FF99 - text mode and then end with $FF9D /$FF9E, since these do the most work. TABLE 9 describes the other $FF9x registers; some of which will be covered in more detail in the GFX tutorial; plus some TEMPS used by SECB.

The previous installment covered the 80 column-attribute mode of the MMU. Now I will explain the NON-attribute text mode, something the service manual did not cover. By setting bit 1 (CRES0) of $FF99 to a '0', the MMU goes into the NON-attribute mode. This means that there is NO attribute byte sent to the screen, that every screen location is for characters, and you are limited to 2 colors. The palette registers involved are $FFB0 for background and $FFB1 for foreground.

Another interesting feature of $FF99 is it's ability to add extra screen lines either with or without attributes. TABLE 10 shows this information. For example: a value of $74 plugged into $FF99 will give you an 80x28 screen with NO attributes. $7D will give you the same WITH attributes.

One interesting thing about the 80x28 screen, is that lines 25-28 will not scroll with the SECB screen routine. SECB knows (in it's code), how many lines to scroll, so it ignores 25-28. To get them to scroll, you would have to change the # of screen lines that it scrolls in SECB's code, or use your own screen routine and scroll it. But even if you don't, this non-scrolling can have some interesting uses! You could take advantage of this feature, and use it for IMPORTANT messages that you want to STAY on the screen. The same is true for the 80x25 screen, line 25 stays put.

LISTING 10 will demonstrate how to set up SECB to use, display, and scroll an 80x28 screen. Type it in, assemble it into memory and run it from Z-BUG. Now, fill the screen with text and count the number of lines on the screen; and they should add up to 28 lines. See, there is nothing to it when you know how! Why didn't TANDY tell us how to do it???

TABLE 9 - other FF9x registers

$FF90	MMU disabled = $CC MMU enabled = $4C
$FF91	00 for Task register - 0 01 for Task register - 1
$FF92 - $FF95	Haven't gotten to these yet!
$FF96 / 97	RESERVED - not used
$FF98	USE - $03 for TEXT USE - $80 for GFX (bit plane) More on this register during the GFX tutorial.
$FF99	SEE Tutorial text
$FF9A	BORDER register (color)
$FF9B	RESERVED - not used
$FF9C	SEE Tutorial text
$FF9D/$FF9E	$D800 = for 80 column text *** SEE Tutorial text $C000 = GFX screen (bit plane) GFX portion will be covered in GFX tutorial.
$FF9F	$00 = NORMAL $80 = 128 column text screen (see text) not very usefull during graphics (see GFX text)

Want to try out a 128 column screen??? Nothing to it!! Just type in LISTING 11, assemble it to memory, and run it from Z-BUG with 'GGO'. It is just a simple program to demonstrate how to set SECB for a 128 column screen. Just start typing anything to the screen, anything at all; a novel, a letter to the editor, anything. When the cursor gets to the far right side of the screen, you will notice that the screen will start to scroll horizontally. If you send a carriage return, or you get to the end of the 128 character line, the screen will jump back to the left margin. You can also move the screen horizontally using the <shift left arrow> or <shift right arrow> keys.

To set this mode, all we did was set BIT 7 of $FF9F to 1 ($80), which is called the "horizontal Virtual Enable" bit. Then by incrementing $FF9F from $80 up (81, 82, 83, etc.) we can move the screen left by one character space. Of course, we also had to change some of the code in SECB

TABLE - 8 Block number VS. screen display offset (see text)

BLOCK	FF9D	9E	BLOCK	FF9D	9E	BLOCK	FF9D	9	BLOCK	FF9D	9E
00	00	00	10	40	00	20	80	00	30	C0	00
01	04	00	11	44	00	21	84	00	31	C4	00
02	08	00	12	48	00	22	88	00	32	C8	00
03	0C	00	13	4C	00	23	8C	00	33	CC	00
04	10	00	14	50	00	24	90	00	34	D0	00
05	14	00	15	54	00	25	94	00	35	D4	00
06	18	00	16	58	00	26	98	00	36	D8	00
07	1C	00	17	5C	00	27	9C	00	37	DC	00
08	20	00	18	60	00	28	A0	00	38	E0	00
09	24	00	19	64	00	29	A4	00	39	E4	00
0A	28	00	1A	68	00	2A	A8	00	3A	E8	00
0B	2C	00	1B	6C	00	2B	AC	00	3B	EC	00
0C	30	00	1C	70	00	2C	B0	00	3C	F0	00
0D	34	00	1D	74	00	2D	B4	00	3D	F4	00
0E	38	00	1E	78	00	2E	B8	00	3E	F8	00
0F	3C	00	1F	7C	00	2F	BC	00	3F	FC	00

to help it along with displaying this mode.

Notice in the listing that we keep a RAM image of $FF9F. This is because $FF9F is a WRITE ONLY register. If you try reading it, all you will get returned is '$1B'. One extra routine was added for SECB to use. It will cause the screen to reset to the left margin when SECB sends a C.R. to the screen. I located it into SECB's copyright area, which is a nice large, unused area of memory for patches. The 128 screen demo just setup an intercept to use this extra code at the start of the routine. The DEMO is fully commented, so you can modify it for other uses.

TABLE 10 - TEXT table information for $FF99

FF99 value	line adjust	end offset	screen size	with attribute	$FF9E adjust	note
$08	$20	$02FF	32x24	NO	$08	
$04	$28	$03BF	40x24	NO	$0A	
$10	$40	$05FF	64x24	NO	$08	
$14	$50	$077F	80x24	NO	$0A	
$1D	$A0	$0EFE	80x24	YES	$14	(1)
$28	$20	$031F	32x25	NO	$08	
$24	$28	$03E7	40x25	NO	$0A	
$30	$40	$063F	64x25	NO	$08	
$34	$50	$07CF	80x25	NO	$0A	
$3D	$A0	$0F9E	80x25	YES	$14	(2)
$68	$20	$037F	32x28	NO	$08	
$64	$28	$045F	40x28	NO	$0A	
$70	$40	$06FF	64x28	NO	$08	
$74	$50	$08BF	80x28	NO	$0A	
$7D	$A0	$117E	80x28	YES	$14	(3)

NOTES:

(1) Normal 80 column screen
(2) Line 25 does not scroll
(3) Lines 25 - 28 do not scroll
(4) screen START = $x000 x = depends upon $FFAx register range that block $36 is mapped into (see TEXT).
(5) screen END = add 'END OFFSET' to screen start for actual end, (lower right corner of screen)
(6) screen without attribute is 2 color only. $FFB0 = background $FFB1 = foreground

The 'line adjust' column in TABLE 10 is used to add to the cursor location so that the cursor can be moved up or down (it is the line length). The 'FF9E adjust' column is used to change where the MMU starts displaying the screen by 1 line (more on this later), so don't confuse the 2 columns. The 'end offset' column is used to calculate the end of the screen, depending on which block the screen is mapped into (remember this from last time?).

For example: if the screen block ($36) is mapped into $FFA3 ($6000-$7FFF) and you chose $3D as the $FF99 value, then the screen would end at $6F9E, ($6000 + $0F9E = $6F9E). If you chose $7D, the screen would end at $717E ($6000 + $117E = $717E), etc..

The best way to see what is happening, is to experiment with some values plugged into $FF99. DISK EDTASM users should set up the 80 column screen by using LISTING 2; and E/A 6309 users will use LISTING 3 to set block $36 into memory for Z-bug to access, just like last time.

Now that you have the screen block in memory, go into Z-BUG and byte mode. Change $FF99 to '08' and you will see that you now have a 32x24 screen with NO attributes. Look carefully at the screen, and you will see that the TEXT is in every other screen location, with strange characters in between. Remember that before you changed $FF99, you were in the attribute mode, so text was written to the screen for that mode.

By changing the value of $FF99, you set the non attribute mode, and the attribute byte is now being displayed as a character! Press the 'clear' key, and start sending ASCII codes to the screen, starting at $6000 with Z-BUG's "slash" command. It will take awhile for the screen to start scrolling, because EDTASM uses SECB's screen routine, which is still set up for 80x24. But you can see that when writing directly to the screen, every memory location is used for characters, while EDTASM is still sending them to every other location (as seen once the scroll catches up). So if you would like to use the NON attribute screen, I would recommend writing your own screen routine.

Now play around with the other '$FF99 values' to see the difference between them; it is quite interesting. Just a reminder, you will be typing BLIND, so type CAREFULLY.

In the 80x25 or 80x28 attribute mode, you can access lines 25 - 28 just like any other line (but not with SECB); write to them directly. Line 25 starts at $6F00, line 26 at $6FA0, 27 at $7040 and 28 at $71E0, with the screen ending at $717E. To return to the REGULAR text screen, just change $FF99 to $1D.

$FF9C seems to be some kind of vertical fine scroll register. Values between 00 and 07 will cause '1/8th' of a screen line to appear, per increment, at the bottom of the screen. A value of '07' will cause line 24 to stay at the bottom of the screen at all times. It will not scroll. It is similar to the 80x25 column screen; but you only have 24 total lines. It too can be useful for important messages that stay on the screen. It's NORMAL value is '00'.

$FF9D / $FF9E is the screen start offset. $FF9D is like a 'coarse' adjustment and $FF9E is like a 'fine' adjustment. During TEXT mode, $FF9D is normally $D8, $FF9E is $00. I have discovered that by adding 04 to the value in $FF9D, you can change what is being displayed by 1 'BLOCK'. For example: if $FF9D/9E is set to $0000, then BLOCK 0 is being displayed. With a value of $0400, block 1 is displayed, and so on (well only the first 24-28 lines worth, depending on $FF99 as above). If you want to use this idea, I would recommend keeping a RAM image of $FF9D/9E, because they are not readable, (similar to $FF91 in part 1). LISTING 9 will demonstrate this shortly.

By adding the value in TABLE 10's "FF9E adjust" column to the value in $FF9D/9E, you can move the display in the 'block' by one LINE. I don't know why the "FF9E adjust" value is so small compared to the normal "line adjust" value, or why some of them are the same; I just found this out by accident. But, it is quite interesting and could possibly have some useful applications. To see what I mean, type in LISTING 9, save it to disk and assemble it to memory. **** WARNING: This routine will WRITE to memory areas used by EDTASM and it's DOS. DO NOT WRITE ANYTHING TO DISK AFTER

TABLE 11 - SECB setup tables (all addresses and data are HEX)

hard	32 COLUMN	40 COLUMN	80 COLUMN	GFX	GFX
$FF90	E032 CC	E03B 4C	E044 4C	E070 4C	E079 4C
$FF98	E033 00	E03C 03	E045 03	E071 80	E07A 80
$FF99	E034 00	E03D 05	E046 15	E072 00	E07B 00
$FF9A	E035 00	E03E 12	E047 12	E073 00	E07C 00
$FF9B	E036 00	E03F 00	E048 00	E074 00	E07D 00
$FF9C	E037 0F	E040 00	E049 00	E075 00	E07E 00
$FF9D	E038 E0	E041 D8	E04A D8	E076 C0	E07F C0
$FF9E	E039 00	E042 00	E04B 00	E077 00	E080 00
$FF9F	E03A 00	E043 00	E04C 00	E078 00	E081 00

RUNNING THIS PROGRAM, or a CRASH IS CERTAIN !!! **** Turn off the computer and reboot before using the disk.

The routine will set up an 80x24 non-attribute screen (for easier reading of text), mark blocks 0 - 59, and start displaying 'BLOCK 00'. It doesn't mark blocks 60 - 63 due to the ROMs being there, and needing the ROM routines (don't want to write over it's code). To 'PAGE' through each BLOCK, use the 'shift' up or down arrows. To page through a particular block LINE by LINE, use the up or down arrows.

When paging line by line into another block, you will notice that the display is 'out of sync' (the start of the next block is not in the far left side). To correct this, pressing the 'CLEAR' key will reset the routine to the start of the LAST block that was set.

When you get to blocks 60 - 63, you can tell that they are the ROMs by looking at the text (you will see the copyright notices). Then the routine will recycle to block 00 again, if you keep going in the same direction.

With some imagination, this DEMO could be put to some useful purpose, it's up to you. If you don't like the screen colors, just change $FFB0 = background $FFB1 = foreground, to the colors of your choice. It's strange how the non-attribute screen uses a background register for a foreground color.

TABLE 8 shows what value to put into $FF9D/9E to start the display with a particular block. Now, if by adding 04 to the value in $FF9D, you change the display by one block; then it goes that by adding 02 to $FF9D, you will change the display start by a half block; 01 by a quarter block, etc. The use of adding 02 to $FF9D was used in PART 2 to set the 2nd half of the screen block (just thought you would like to know).

You now have plenty to experiment with until the next installment. Along with PARTS 1 and 2, let your imagination go to work! Next time, I will cover what I have found with graphics and more on the $FF9x registers.

LISTING 9 $FF9x Demo program

```
GO  NOP
  LDD   #$3030  ###
  STD   NUM     ### set block # to '00'
  LDA   #$14    = 80x24 NON-attribute mode
  STA   $FF99   set it
  LBSR  SETUP   now go mark the blocks
  LDD   #$0000  ### start with block '00'
STORE  STD   TEMP
                save block number offset
NEXT  LDD TEMP  get current block # offset
  STA $FF9D     ** set
  STB $FF9E     ** offset now
  JSR $A1C1     scan keyboard
  CMPA #$0A     down arrow ?
  BEQ DOWN      yes, scroll screen down 1 line
  CMPA #$5E     up arrow ?
  BEQ  UP       yes, scroll screen up 1 line
  CMPA #$5F     'shift' up arrow ?
  BEQ  MUP      yes, 'page' to next block
  CMPA #$5B     'shift' down arrow ?
  BEQ  MDOWN    yes, 'page' to previous
                block
  CMPA #$0C     'clear' key ?
  BEQ  RESET    yes, re-sync display
  CMPA #3       'break' key ?
  BNE  NEXT     no, ignore all other keys
  LDD  #$D800   address of standard screen
  STA  $FF9D    reset MMU for standard
  STB  $FF9E    text screen
  LDA  #$1D     standard value for $FF99
  STA  $FF99    set it now
  SWI  EXIT     to Z-BUG
DOWN  LDD  TEMP get current offset
  SUBD #$0A     bump to previous line
  BRA  STORE    now go do it
UP  LDD TEMP    get current offset
  ADDD #$0A     bump to next line
  BRA  STORE    now go do it
MDOWN LDD TEMP  get current offset
  SUBA #4       bump to previous page
  STD  BLKSET   save for reset routine
  BRA  STORE    go do it
MUP  LDD TEMP   get current offset
  ADDA #4       bump to next page
  STD  BLKSET   save for reset routine
  BRA  STORE    go do it
STRING  LDA  ,X+  get next character
  PSHS A        save for stop test
  ANDA #$7F     drop MSB
  STA  , Y+     send it to screen
  TST  , S+     was last character a 'stop' ?
  BPL  STRING   no, then loop for more
  RTS           return
SETUP  LDA #$FF  ** get ready
  STA  BLOCK ** for block '00'
SETNXT LDA BLOCK get last block set
  INCA  bump it by 1
  STA  BLOCK    save current block number
  CMPA #$3B     last block to mark ?
  BHI  STOP     yes, then stop
  STA  $FFA3    set block
                ** $FFAB for E/A 6309 users
  LDY  #$6000   = screen start
  LEAX TXT2,PCR get text to mark block
  LBSR STRING   go write it into block
  LEAX NUM,PCR  get block number
  LBSR STRING   go write it into block
  LDD  NUM      get current block number
  ADDB #1       bump it by 1
  CMPB #$39     is 'units' higher than a 9 ?
  BHI  BUMP     then fix to keep DECIMAL
  STD  NUM      save current block number
  BRA  SETNXT   go set next block
BUMP  LDB  #$30 reset 'units' to '0'
  ADDA #1       bump 'tens' by 1
  STD  NUM save current block number
  BRA  SETNXT   go mark next block
STOP  LDA #$3B  = original block that
                needs to be put back
  STA $FFA3 put it back
                ** $FFAB for E/A 6309 users
  RTS           done - RETURN
TXT2  FCC / START OF BLOCK # /   ## -
(note spaces)
  FCB $A0  = space + stop code
NUM  FDB  $3030 block number (decimal)
                storage
  FDB $2020  = spaces
  FDB $2020  = spaces
  FCB $A0  = space + stop code
BLOCK  RMB1  = block number (hex) storage
BLKSET RMB2  = 'reset' routine storage
TEMP   RMB2  = 'offset' storage
    END
```

LISTING 10 - 80x28 column demo

```
  SECB = Super Extended Color Basic
GO  ORCC  #$50  disable interupts
  CLR $FF91  set TR=0
  LDD #$3180 = last screen line
  STD $F688  set last screen line in SECB
  LDD #$30E0= line 27
  STD $F875  set it for SECB scroll routine
  LDD #$1C1B  $1C = 27 $1B = 28
  STA $F683  set SECB
  STB $F87F  set SECB
  JSR $F679  do SECB's 80 column setup
             and clear screen
  LDB #1     ***
  STB $FF91  *** set TR=1
  LDA #$7D   = 80x28 column with
             attribute code
  STA $FF99  set it
  ANDCC  #$AF  enable interupts
  SWI    EXIT  back to Z-BUG
  END
```

LISTING 11 - 128 column demo

```
*** This portion sets up SECB for this routine ***

GO  ORCC #$50 disable interupts
  CLR $FF91  set TR=0
  LDD #$7EF7
  STD $F824  ***
  LDA #2
  STA $F826  *** set intercept here
  LDD #$3800 = screen end address
  STD $F688  set it for SECB
  LDD #$3700 = last screen line start
  STD $F875  set it for SECB
  LDD #$0140 = 128 character count
             (128+128 for attributes)
  STD $F870  set it for SECB
  LDA #$80   = max. characters per line
  STA $F681  set it for SECB
  JSR $F679  set SECB 80 column screen,
             clear screen
  LBSR  MOVE  move <C.R.> intercept here
              for SECB to use
  LDB #1     ##
  STB $FF91  ## set TR=1
  LDA #$80   ***
  STA $FF9F  *** set $FF9F for 128 column
  STA TMP9F set RAM image of $FF9F
  ANDCC  #$AF  disable interupts

*** now data is entered thru the keyboard and
displayed to the screen

POLL  CLR $FF91  set TR=0 for ROM use
  JSR $A1B1  poll keyboard
  LDB #1     ##
  STB $FF91  ## set TR=1 for this routine
  BEQ POLL   loop if no key pressed
  CMPA #$15  <shift right arrow> key?
  BEQ BACK   yes, do scroll left routine
  CMPA #$5D  <shift left arrow> key?
  BEQ FORWRD  yes, do scroll right routine
  CMPA #3    <break> key?
  BEQ  BREAKyes, exit this routine
```

```
  CMPA  #$0D   <enter> key?
  BEQ ENTER    yes, do 'C.R.' routine
  BSR DISPLA   display any other key pressed
  BSR   TEST   check to see if cursor at far
               right of screen
  BRA POLL  go for more entries
  DISPLACLR $FF91   set TR=0
  JSR   [$A002] display character
  LDB  #1      ##
  STB  $FF91  ## set TR=1
  RTS          return

** see if 79 characters have been displayed.
   If so, start scrolling the screen left until 128
   have been displayed.

TEST LDB $FE02get SECB's character count
  CMPB  #$4F   displayed 79 characters yet?
  BLS  D1      no, then exit
  LDA  TMP9F   get image of $FF9F
  INCA         bump it for scroll left
  STA  TMP9F   save new value
  STA  $FF9F   cause scroll left by one
               character
D1   RTS       return

** The <shift left arrow> key will cause screen to
   scroll left no matter how many characters are
   displayed.

BACK  LDB TMP9F  get RAM image of $FF9F
  CMPB  #$B3   ##
  BHI   B1     ## don't allow a scroll further
               than this
  INCB         bump it for scroll left
  STB  TMP9F   save new value
  STB  $FF9F   cause a scroll left
B1   BRA POLL  back to keyboard

** The <shift right arrow> key will cause screen
   to scroll right

FORWRD   LDB TMP9F   get RAM image
                     of $FF9F
  CMPB  #$80   ##
  BLS   F1     ## scroll no further that this
  DECB         drop count by one
  STB   TMP9F save new value
  STB   $FF9F cause scroll right by
              one character
F1   BRA POLL  back to keyboard

** The <enter> key will cause the screen to reset
   to the start (on the left margin)

ENTER   LDB #$80   start value for $FF9F
  STB  TMP9F save new value
  STB  $FF9F  reset screen to left margin
  BSR  DISPLA  display last character
  BRA  POLL  back to keyboard

** Reset SECB to the normal 80 column screen
   and exit to Z-BUG.

BREAK   CLR $FF9F  set $FF9F back to
                   80 columns
  ORCC  #$50   disable interupts
  CLR   $FF91  set TR=0 to access ROMs
  LDD   #$2E60 = last screen line start
  STD   $F875  set it into SECB
  LDD   #$2F00= screen end address
  STD   $F688  set it for SECB
  LDD   #$00A0= max. characters per line
               for scroll
  STD   $F870  set it for SECB
  LDA  #$50    =max characters per line
  STA  $F681   set it for SECB
  JSR  $F679   do 80 column setup
  LDA  #1      ##
  STA  $FF91   ##set TR=1 for this routine
  ANDCC  #$AF  disable interupts
  SWI          return to Z-BUG

** This routine just moves 'DATA', so SECB can
use it.

MOVE  LDX #$F702= destination of routine
  LEAYDATA,PCR  = source of routine
  LDB  #$10
NXT LDA   ,Y+    ***
  STA    ,X+   *** move routine to new location
  DECB         ***
  BNE    NXT ***
  RTS        done, return

** This routine will reset the screen to the left
   margin when SECB sends a C.R. at the end
   of a line. The intercept was set at the start of
   this program.

DATA   PSHS  B  save 'B'
  LDB  #$80    = value to reset $FF9F
  STB  TMP9F save new value in RAM image
  STB  $FF9F   set $FF9F to new value
  CLRA         set up for original routine
  PULS  B      get this register
  JMP  $F802   back to original routine
TMP 9FRMB 1 = RAM image of $FF9F register
  END
```

68K Performance Guide

continued from page 13

12. On original 68000-based cores it makes sense to exchange any shift larger than 16 bits with a swap, a smaller shift and maybe a clear. For example:

```
    lsr.l #18,Dn
becomes
    clr.w Dn ; leave out if you don't mind a mess in the upper word
    swap Dn
    lsr.l #2,Dn
```

IV. PROBLEMATIC

These are some hints that reduce code readability, portability or have harmful side effects:

1. and's and or's (even immediate versions) are very often faster than bit sets/clears.

2. Using Booth's algorithm for multiplication by a constant will often be much faster than mul* instruction, but what do you do if this constant changes? You might be able to write a macro that generates Booth sequence for a given constant if your assembler has a powerful enough macro processor. BTW, mul* instructions always affect the entire destination register regardless of the operation size. Use the slower long version only if you really have to. Try to express a division by a constant as a multiplication by a constant and division by a constant power of two (shift right, see also III.12). For example, a division by 12 can be expressed as a multiplication by 85 and division by 1024 (shift by 10):

```
    move.l Dn,Dm
    lsl.l  #2,Dm   ; make this 2 adds on anything better than original 68000-based core. (see III.9)
    add.l  Dm,Dn   ; Dn = Dn * 5
    move.l Dn,Dm
    lsl.l  #4,Dm
    add.l  Dm,Dn   ; Dn = Dn * (5 + 5 * 16) = Dn * 85
    lsr.l  #10,Dn  ; Dn = Dn * 85 / 1024
```

3. On original 68000 and derivative cores, 'moveq #0,Rn' is faster than clr. How ugly.

4. Don't use any 'non-quick' immediate instructions in a loop, try to pre-load everything into a register.

5. Fast CPU32 polling:

```
    moveq #-1,Dn
    moveq #READYBIT,Dm ; static btst can't be loop-moded, see III.7
poll:
    btst   Dm,some_memory-mapped_device_register
    dbne  Dn,poll
```

of course, using III.2 we can speed it up even more if the signalling bit happens to be bit 7,15 or 31:

```
    moveq #-1,Dn
poll:
    tst.x   some_memory-mapped_device_register
    dbmi  Dn,poll
```

This is very fast because the loop mode is utilized and dbcc is perfect for overlapping access to the device register which is probably even slower than regular memory. The problem is that it will fail if Dn ever wraps around (after 65K iterations).

V. NEEDED

So, which CPU32 instructions have tails? I understand any access to memory, but not the source in move, but I am not sure. If you have any additions, e-mail me or write this magazine.

ADVERTISER'S INDEX