Wheels 64/128 Programming Notes

Below are a collecting of emails send to me back when Wheels 64/128 for Commodore GEOS users was relatively new, and many GEOS programmers were trying to get a grasp on it. Maurice Randall, the author of Wheels, was extremely helpful with the questions we had, and often shared other emails he'd written to other people with me because he knew the content was of interest.

These emails were later printer out and bound for my reference, but I no longer have the original digital form. What you see below are digital scans of that bound booklet.

If you think you can OCR it and send me a copy, it would be appreciated, but I'm not optimistic. Many of those printouts were in awful shape, and the low quality of the scans reflect this.

Anyway, I hope these notes are as helpful to you as they were to me.

All the thanks, of course, goes to Maurice for composing his developers guide one email at a time. :)

– Bo Zimmerman
3/19/2016

Status: Read
Received: from bos1b.delphi.com (bos1b.delphi.com [199.93.4.2]) by mail
Received: from delphi.com by delphi.com (PMDF V5.1-8 #23839)
 id <01J53VILZ9PS90YP09@delphi.com> for bo@zimmers.net; Tue,
 8 Dec 1998 20:11:56 EST
Date: Tue, 08 Dec 1998 20:11:56 -0500 (EST)
From: ARCA93@delphi.com
Subject: Re: Wheels 128 testing
To: bo@zimmers.net
Message-id: <01J53VILZ9PU90YP09@delphi.com>
X-VMS-To: IN%"bo@zimmers.net"
MIME-version: 1.0
Content-type: TEXT/PLAIN; CHARSET=US-ASCII

Bo,

There's a way you can cheat a little if you want.

Make sure you call OpenDisk at least once, most likely at the
start of what you're doing.

Since what you're doing is switching back and forth between disk
drives, you normally need to call OpenDisk again each time you
access the disk. But if all you need to do is load in the BAM,
you can cheat.

```
openError==$907a

 LoadB openError,#0
 jsr GetDirHead
```

If you don't need the BAM read, then skip the GetDirHead.
Use this only for a highly specialized drive routine, such as
what you're doing. Normal file accesses should use OpenDisk.
Setting openError to zero will make the driver think it has
successfully opened the disk the last time OpenDisk was called.
So, it's up to your program to make sure there's a good disk
in the drive and it's formatted.

This should help speed things up and limit the amount of head
movement.

Keep in mind, this will only work in Wheels. This variable is
not used in GEOS. So, you need to make sure Wheels is running.

```
version==$c00f
driverVersion==$904f

 lda version
 cmp #$40
 bcc 90$  ;branch if not Wheels
 lda driverVersion
 cmp #$50
 bcc 90$  ;extra verification check, branch if not Wheels.
;at this point it's Wheels.

 ...

90$
;at this point, it's not Wheels.
```

The driverVersion byte is in every Wheels disk driver. If it's $50 (V5.0) or higher, then it's a Wheels disk driver. This byte is not valid in GEOS, but the byte that's there is always less than $50. That's why I chose to start the Wheels drivers out at V5.0.

-Maurice

I'll take a look at the macro part.

I put the info on copying in the Delphi forum for the benefit of others, and the following is the text in that message:
==========

COPYING FILES and CHANGING PARTITIONS
From a programmer's point of view.

Copying files in Wheels is pretty simple. Here's how to do it:

First of all, the currently active directory is always the source directory. So open the correct drive and subdirectory first.

Now, set up the following:

r0 - points to a null terminated filename for the destination name. This can also be used to duplicate a file. If the source and destination directories are the same and r0 points to a filename that is different from the source filename, then a file duplication will take place within the same directory.
If the names are different and the source and destination directories are also different, then the file will be copied and the copy will receive a new name. Whatever r0 points to is what the destination file will be named.

dirEntryBuf is loaded with the dir entry of the source file.

r3L - bits 0-5 contains the destination drive number (8-11).
- bit 7 set copies the file to the system directory.
clear copies the file into the main directory.
- bit 6 set invokes the single drive copier and the user will be prompted to insert the source and destination disks as needed.

r2L - If the destination is a partitionable device, this is loaded with the destination partition number. It's safe to set this even on non-partitionable devices such as the 1541. Therefore, it's not necessary to determine the drive type prior to copying a file.

r1L,r1H - track and sector of the destination directory on a native mode partition or ramdisk. These values are also meaningless on a 1541,71, or 81 type directory.

If the destination turns out to be the system directory of the main directory, then a simple directory entry "move" will take place.

r3H - set bit 7 to force a "move" instead of a copy, if desired, provided the destination is within the same partition as the source. If this is not the case, then a copy is performed instead of a move.

r2H - bit 7 if clear, will replace the file on the destination if one of the same name as what r0 points to exists.
if set, then the file will be skipped if one of the same

name is found.

RETURNS:

x - equals 0, if no error. The copy was successful.
    255 means a file of the same name existed if bit 7 of r2H
      was set. The copy did not proceed.
    Any other value indicates an error and no copy was performed.
r3H - indicates if a move or copy was performed. Bit 7 set indicates
      a move and clear indicates a copy. Remember, the copier can
      override the choice of move or copy when necessary.
The source directory will be opened upon return.

DESTROYS:
$7900-$7fff is used by the filecopier. If you need this area, you
must save it prior to the copy and restore it afterwards.

Invoking the copier is rather simple. But first you must set up
the above registers.

Here's an example:
Let's say the destination will be partition 5 of drive C. And
the desired subdirectory is the currently open directory and
we're currently in that directory.
The source will be drive A which is a 1541.

```
dirHeadTrack==$905c    ;the current directory header track.
dirHeadSector==$905d   ;the current directory header sector.
GetHeadTS==$9063       ;returns the current partition number in r2L.
GetNewKernal==$9d80    ;calls in a desired kernal group.

Copy1File:
  PushB curDrive
  jsr OpenDisk
  PushB r1L
  PushB r1H
  jsr GetHeadTS          ;get the current partition number into r2L
  PushB r2L
  lda #8
  jsr SetDevice
  jsr OpenDisk
  LoadW r6,#fNameBuffer
  jsr FindFile           ;load dirEntryBuf.
  LoadW r0,#fNameBuffer ;destination name.
  PopB r2L               ;destination partition.
  PopB r1H               ;destination dir sector.
  PopB r1L               ;destination dir track.
  PopB r3L               ;destination drive.
  lda #9                 ;kernal group 9.
  jmp GetNewKernal       ;run the first routine in group 9.

fNameBuffer:
  .byte "filename",0,0,0,0,0,0,0,0,0 ;enough for 16 chars plus null.
```

NOTE: the above routine didn't do any error checking in order
to keep the routine easy to follow. Be sure to add error checking
after the calls to routines such as SetDevice, OpenDisk, and FindFile.

In the above example, there is no need to restore the kernal
since bit 6 of the accumulator was clear upon calling GetNewKernal.

It will be restored upon completion of the routine. This method allows you to call GetNewKernal from anywhere in normal memory. If you decide to leave the kernal in memory, it will occupy $5000-$5fff and you can then call the first routine which is "CopyFile" at $5000 repeatedly with each file you wish to copy. However, you must remember to not try to access your own code in this area because it won't be there.

If you wish to use this method, then the only change in the above example is near the end of the routine:

```
RstrKernal==$9d83
CopyFile==$5000
  ...
  lda #(9|64)
  jsr GetNewKernal
  jsr CopyFile
  ...

;do whatever you need in here.

  ...
  jmp RstrKernal    ;this only trashes the accumulator.
```

By setting bit 6 of the accumulator, the memory at $5000-$5fff will be saved and the kernal brought in, but nothing will be run. The above example runs the copier by calling CopyFile. When RstrKernal is called, the area at $5000-$5fff will be restored.

More stuff:

To open any directory on a native partition, load r1L,r1H with the track and sector of the subdir and call OpenDirectory. This does basically the same thing as OpenDisk.

```
OpenDirectory==$9053
```

Here's some of the routines that are in group 5:

```
ChgParType==$5000
ChPartition==$5003
ChSubdir==$5006
ChDiskDirectory==$5009
TopDirectory==$500f
UpDirectory==$5012
DownDirectory==$5015
GoPartition==$5018
ChPartOnly==$501e
FindRamLink==$5027
```

If your application includes a dialogue box with the "DISK" icon, that's all you really need to let the user select any partition or subdirectory on a CMD device or a subdirectory on a native ramdisk. But the above routines are also provided if you wish to add additional partition and directory capability.

Here's a very brief description of the above routines:

ChgParType

Call this with r4L holding either a 1 for a native type or a 4 for a 1581 type, and the appropriate driver will be invoked for this CMD device.

It's rare that this routine is ever needed. It's mainly used by the operating system when switching partitions.

ChPartition

This will call up a system dialogue box, allowing the user to select a different partition or subdirectory. This starts out by displaying a list of the currently available partitions.

This may not be called from within another dialogue box unless the programmer is familiar with how to preserve dialogue box variables.

ChSubdir

This is similar to ChPartition, except that it starts out by displaying a list of the subdirectories within the current directory. The user is also given the ability to change partitions.

This may not be called from within another dialogue box unless the programmer is familiar with how to preserve dialogue box variables.

ChDiskDirectory

This routine may be safely called from within another dialogue box.

This works just like ChPartition and ChSubdir other than the ability to call it from a dialogue box. It will start the user out in the appropriate mode.

TopDirectory

This will open the root directory of the current drive if it's a native mode partition or native ramdisk. If a real drive or the RamLink, then the DOS in the device is also correctly pointed to the root directory.

UpDirectory

This will open the parent directory of the current drive if it's a native mode partition or native ramdisk. If a real drive or the RamLink, then the DOS in the device is also correctly pointed to the root directory.

DownDirectory

This will open a subdirectory within the current directory if it's a native mode partition or native ramdisk. If a real drive or the RamLink, then the DOS in the device is also correctly pointed to the root directory.

Call this with dirEntryBuf containing the directory entry of the desired subdirectory.

GoPartition

This will select a partition on a CMD device. Call this with .x holding the number of the desired partition. The partition must

be either a 1581 or native mode type. The correct driver will be installed by this routine and the current directory on the desired partition will be opened. The directory is whichever one is listed by the drive's own DOS as the current directory.

ChPartOnly

This is just like ChPartition, except that it doesn't allow the user the ability to change subdirectories. Only a partition can be selected. All other aspects are the same as ChPartition.

FindRamLink

This will search for a RamLink. If found, .x will hold the "real" device number of the RamLink, not the drive letter assignment as seen by the user. This allows the programmer to address the RamLink through direct DOS calls if needed. If x=0, then there is no RamLink on the system.
This routine works whether the RamLink is configured for use by the operating system or not.

Here's a couple of small examples you can include in your programs:

```
;this will find the RamLink. Upon return, .x can be
;tested.
WhereIsRamLink:
  lda #(5|64)
  jsr GetNewKernal
  jsr FindRamLink
  jmp RstrKernal

;this will pop up a dialogue box allowing the user to
;select a partition or subdirectory.
GetNewDirectory:
  lda #(5|64)
  jsr GetNewKernal
  jsr ChDiskDirectory
  jsr RstrKernal
  jsr GetHeadTS
  MoveB r2L,thisPartition  ;save the user's choice of partitions.
  MoveB r1L,thisTrack      ;save the header track.
  MoveB r1H,thisSector     ;save the header sector.
  rts

thisPartition:
  .block 1
thisTrack:
  .block 1
thisSector:
  .block 1
```

In this last example, you can load r1L and r1H with the header track and sector prior to calling OpenDirectory if you decide to reopen this directory. OpenDirectory is safe to call on any device. If you load bad values when calling OpenDirectory on a 1541 or 1581 or anything that doesn't support subdirectories, it won't hurt anything. The driver itself will substitute the correct track and sector value without causing an error.

-Todd Elliott

Reply

Forward

Top of Page
Message 48925
Previous
From:
(ARCA93)
To:
Todd Elliott (EYETH)
4 of 4
Posted:
12/27/98 7:00 PM
Reply to:
48921

Right Todd,
The application currently running always has some ram available for use. Originally, when I first released Wheels 64, the amount of available ram was about 30K. This was from $0000-$78ff in the firt bank (bank 0) of the REU. But now, I'm allowing more than that for the applications to use. They have a little over 32K for use now. This would be all the way up to $82ff. The OS doesn't use any of this part of bank 0.
This is how the Dashboard and the new Toolbox (128 version as well as the new 64 version) are able to completely load into memory. The Dashboard 128 is 40K and Toolbox 128 is about 54K. Of course, they are not running at the same time, but when the Dashboard is on the screen, it is considered to be the currently running application, and so it can make use of this extra ram without having to use any other bank.
-Maurice

Reply

Forward

Wheels programming

Message 48916

From:
(ARCA93)
To:
ALL
1 of 4
Posted:
12/26/98 7:43 PM
Reply to:
New Thread
Next

Wheels 64 and 128 programmers,
Here's some prelimary (and brief) info on how to allocate ram from the REU device for use by a Wheels application.
First, some equates and symbol definitions:
GetNewKernal==$9d80 RstrKernal==$9d83
GetRAMBam==$5000 PutRAMBam==$5003 AllocAllRAM==$5006
AllocRAMBlock==$5009 FreeRAMBlock==$500c GetRAMInfo==$500f
RamBlkAlloc==$5012 RemoveDrive==$5015 SvRamDevice==$5018
DelRamDevice==$501b RamDevInfo==$501e
The ram handling routines reside within the "extended kernal" which is stored in the last bank of the REU. Only a small portion of the extended kernal is used for the ram handler. There is much more in the extended kernal than this. But for this discussion, we will talk only about the ram routines and how to access them.
To bring the extended kernal into memory requires calling the routine "GetNewKernal". But you must also indicate which portion of the extended kernal you wish to access. The ram routines are contained in what is known as Group 0. You load the accumulator with a 0 in the lower nybble and also set bit 6 and then call GetNewKernal as follows:

```
  lda #(0|64)
  jsr GetNewKernal
```

What this does, is it will swap the memory at $5000-$5fff with that of the area of the REU that holds the ram handling routines. By setting bit 6, you are telling GetNewKernal to swap the memory but don't run any of the code. If bit 6 was clear, then the first routine in this group would be executed and upon termination, the memory would be swapped back. But we want to use more than the first routine, so we'll want to leave the code in CPU memory until we're done with it.
Now, that the ram handler is in memory, we want to find out if there is any ram available for use. Ram from the REU device must be allocated in chunks of 64K at a time. Even if your program only needs 1 byte, you have to allocate a whole 64K bank. Let's see how we can find out how much ram is available.

```
  jsr GetRAMInfo
```

It's as simple as that. Now, all you have to do is check a few registers.
r4H contains the total number of free 64K banks. r2L contains the total number of consecutive free 64K banks. r3L contains the starting bank of the largest free area.
At this point, you can choose to allocate the ram or just go ahead and use it. Be aware that if you don't allocate it and your program uses a desk accessory that can also allocate ram, it might step all over your ram bank. But if you're just going to use the ram and be done with it and then exit, don't worry about allocating it, because you will have to also deallocate it upon exit. Or the ram won't be available for use

by other applications.
But in either case, you already know that r3L contains a bank number that is available for use, provided r2L is a non-zero number. If r2L=0, then no ram is available and you should inform the user with a message dialogue box.
Now, when it comes to allocating ram, you can allocate 1 or more banks at a time. You can choose to just allocate the ram and deallocate it upon exiting your application. Or you can add your ram bank(s) to the partition table. This provides an added safeguard in case the partition table is ever validated to free up allocated yet unused ram banks. Having your ram listed within the partition table is really only necessary if you wish to keep your ram banks allocated during a lengthy computing session, such as what the Toolbox does when it assigns ram banks to a ramdisk.
By placing your ram allocation in the partition table, you can always reuse the same banks each time the user runs your application. This is purely a matter of choice and also depends on the design of your application and whether or not you will need to allocate ram and/or store its listing in the partition table.
If you only want to allocate ram, load r2L with the number of 64K banks needed. Load r3L with the desired starting bank or 0 if you wish to let the OS assign a starting bank. And then call RamBlkAlloc.

```
  LoadB r2L,#1 ;we need one bank of ram.
  LoadB r3L,#0 ;let the OS pick a bank for us.
  jsr RamBlkAlloc
```

If the ram allocation was successful, .x will equal 0, otherwise the bank that was just allocated.
Now, this same bank can be used with the normal StashRAM, FetchRAM, and SwapRAM routines which are in the standard kernal. Now that we've allocated our ram, let's put the extended kernal back and continue. This is simple:

```
  jsr RstrKernal
```

So, here it is again for an application that just needs to allocate one bank of ram:

```
Get1Bank:
  lda #(0|64) ;swap group 0
  jsr GetNewKernal ;into memory.
  LoadB r2L,#1 ;we want one bank.
  LoadB r3L,#0 ;and don't care where it begins.
  jsr RamBlkAlloc ;get a bank of ram.
  jmp RstrKernal ;.x is preserved.
```

Put the above routine in your source code and call it when you need to allocate a bank of ram for your own use, like so:

```
  jsr Get1Bank
  txa
  bne 90$
  MoveB r3L,ourOwnBank
  rts
 90$
  ... display an error message here
ourOwnBank:
  .block 1
```

See how easy it is?
Now, how do we free up this bank of ram when we're ready to quit our application?

```
FreeOurBank:
  lda #(0|64)
  jsr GetNewKernal ;get group 0 into memory.
  jsr GetRAMBam ;get the ram bam into the workspace.
  MoveB ourOwnBank,r6L ;indicate which bank to free up.
```

```
  jsr FreeRAMBlock ;and then free it up.
  jsr PutRAMBam ;write the modified ram bam.
  jmp RstrKernal ;and restore the kernal and memory.
```

As you can see, it's a little different to free up one bank at a time. If we had listed our ram in a partition table, we could have sent a request to free up all the ram banks listed in our partition, but this example isn't doing that, so we must do it as in the above routine. One thing you must know. The code you're using to call the extended kernal can't reside within $5000-$5fff, because your code will get swapped out when the extended kernal is brought in.

That's all there is to it when we need a bank of ram. Now how do we access the bank once we've got one to use? Simple, we just use the same routines that have always been available since GEOS 2.0. These routines are in the standard kernal and don't need anything special to access them. In fact, you can call them from almost any memory configuration. It doesn't matter if you've already called InitForIO or not, you always have access to these ram access routines. They are as follows:

StashRAM==$c2c8 FetchRAM==$c2cb SwapRAM==$c2ce VerifyRAM==$c2d1 DoRAMOp==$c2d4

It's easy to figure out what the above routines do by their names, except for the last one. However, DoRAMOp is the actual workhorse of the whole bunch. Each of the first four routines will call DoRAMOp to do the actual work.

In each case, you will load the following registers:

r0 holds the CPU address. r1 holds the REU address. r2 holds the number of bytes r3L holds the REU bank.

So, if we want to stash 50 bytes of ram located at $2000 to the beginning of our bank of ram in the REU, the following will get the job done:

```
  LoadW r0,#$2000
  LoadW r1,#$0000
  LoadW r2,#50
  MoveB ourRamBank,r3L
  jsr StashRAM
```

Not too hard, huh?

If you want to bring those same 50 bytes back into the computer, just use FetchRAM instead of StashRAM. If you want to swap the 50 bytes of computer memory with the 50 bytes of REU memory, just use SwapRAM.

So, what does DoRAMOp do? Actually, StashRAM, FetchRAM, SwapRAM, and VerifyRAM only load .y with a value. DoRAMOp performs the desired job according to the value in .y. So, you could also do the same jobs by loading .y with a value and calling DoRAMOp to do the job. Check this out:

```
STASH=$90 FETCH=$91
StashBytes:
  ldy #STASH
  .byte 44 FetchBytes:
  ldy #FETCH
  LoadW r0,#$2000
  LoadW r1,#$0000
  LoadW r2,#50
  MoveB ourRamBank,r3L
  jmp DoRAMOp
```

See how you can make your code shorter by combining two routines? Of course this example is only good if you're always moving the same bytes to/from the same locations.

An added benefit of using these system routines is that you, as a programmer, don't even need to deal with the type of REU being used. All these routines work just as well with a 1750, geoRAM, RamLink

DACC, and the SuperRAM. Any ram device used in Wheels can use these routines.
Before long, I hope to have available "A Thumber's Guide to Wheels Programming".
Until then, all you have to do is ask for the info and I will respond as I did here.
-Maurice
p.s. See how easy it would be to patch up geoCanvas for Wheels?
Reply

Forward

Top of Page
Message 48917
Previous
From:
CS (CINDYSIMMS)
To:
(ARCA93)
2 of 4
Posted:
12/27/98 9:54 AM
Reply to:
48916
Next

I think I just read "Everything you wanted to know about REUs but did not know how to ask;)". Looking forward, very much, to the Thumbers guide to Wheels Porgramming. Your artilce and examples are so much clearer than the guides that Berkly did for geos. Thanks for sharing.
Reply

Forward

Top of Page
Message 48921
Previous
From:
Todd Elliott (EYETH)
To:
CS (CINDYSIMMS)
3 of 4
Posted:
12/27/98 12:37 PM
Reply to:
48917
Next

Maurice-
As I understand it, there is always 30Kb or something near that figure which is always allocated for the program's free use under Wheels. An application does not need to call any ram allocation/deallocation routines to use this free area, right? And I just simply load the first bank number (zero) for the regular GEOS routines such as StashRAM, etc. and access this area?
Of course, if an application needs more than 30Kb, it can use Wheel's dedicated RAM routines as described earlier.

geoBeap D64 format program

Beginning Message 49020 Previous
From: Bo Zimmerman (BOZAC) To: CS (CINDYSIMMS) 3 of 6
Posted: 1/2/99 1:05 AM Reply to: 45908 Next
Hi Cindy. I was actually made aware of this by Fender and co. back in 1
and wrote a "fix". The problem, you see, is that without an REU, GEOS c
go back and forth between different device drivers. Since you are likel
trying to de-D64 a file from a 1581 disk to a 1541 disk, the problem co
up. The "fix" I put in was to have the program check numDrives directly
instead of peering at the driveType table. This reveals to geoBEAP that
1 viable drive (the boot drive) is available, and informs the user of t
I've already gotten word from Jeff Jones that he'll publish the fix, bu
waiting to get my Wheels changes in first. A daunting task, it turned o
Also, the new geoBEAP will support CMD Native drive archives in .BEP fo
and will run in GEOS 128 in 80 columns. This will be the only program I
ever ever make THAT change to again. I'll post something when I get tho
sent in. - Bo

Reply Forward Delete

Message 4 of 6 was Deleted

Top of Page Message 49030 Previous
From: (ARCA93) To: Bo Zimmerman (BOZAC) UNREAD 5 of 6
Posted: 1/2/99 12:42 PM Reply to: 49020 Next
To all GEOS/Wheels programmers,

Here's how I'm dealing with the 40 and 80 column screens now, in the programs that must deal with both.

Here's an equate:

```
c128Flag=$c013 ;bit seven set indicates 128 mode.
```

Here's two variables that I set up:

```
dblB:
  .block 1 add1W:
  .block 1
```

And here's a routine that I use very early in my program:

```
SetDblBits:
  lda c128Flag
  and #%10000000
  sta dblB ;bit seven set if 128 mode.
  lsr a
  lsr a
  ora dblB
  sta add1W ;bits 5 and 7 set if 128 mode.
  rts
```

Now, I have two variables that can be used for doubling in all modes. T
can be used in GEOS 64 and Wheels 64 and will have no effect on the
registers you use them on. But in GEOS 128 and Wheels 128, they will do
the values in the registers if in 80 column mode. Here's how I use them
have three different routines I can call.

```
AdjPixels:
  lda r3H
  ora dblB
  sta r3H
  lda r4H
  ora add1W
  sta r4H
  rts

AdjCards:
  lda r1L
  ora dblB
  sta r1L
  lda r2L
  ora dblB
  sta r2L
  rts

AdjR11:
  bit c128Flag ;64 mode?
  bpl 10$ ;branch if so.
  bit graphicsMode ;are we in 80 column mode?
  bpl 10$ ;branch if not.
  asl r11L ;double r11.
  rol r11H
 10$
  rts
```

Now, anytime I need to adjust any screen coordinates, I can call the appropriate routine. For instance, I want to place some text on the scr 50 pixels from the left and 100 pixels down.

```
  ...
  LoadW r0,#testTxt ;point to the string.
  LoadW r11,#50 ;left coordinate.
  LoadB r1H,#100 ;baseline coordinate.
  jsr AdjR11
  jsr PutString ;display the string to the screen.
  ...

testTxt:
  .byte "This is a test string.",0
```

This string will always appear in the same relative position on the scr no matter which mode we are running in. We can do the same thing with a rectangle:

```
  ...
  LoadB r2L,#50 ;top coordinate.
  LoadB r2H,#100 ;bottom coordinate.
  LaodW r3,#80 ;left coordinate.
  LoadW r4,#239 ;right coordinate.
  jsr AdjPixels
  jsr Rectangle ;draw a filled rectangle.
```

If this routine is used in 80 column mode, it will draw a rectangle at left coordinate of 160 and a right coordinate of 479. Notice that the r coordinate didn't get doubled? Instead it got doubled and 1 more added it. AdjPixels sets bits 13 and 15 (bits 5 and 7 of the high byte) in r4 This tells the 128 kernal to double the value and add one more. On the

hand, AdjPixels only sets bit 15 (bit 7 of the high byte) in r3. This doubles the left coordinate.

Why do we add 1 to the right hand value? If we didn't we wouldn't be exactly true in our scaling. Imagine that you used Rectangle to clear t screen. If you only double the right hand value from 319 to 638, you wouldn't be reaching the far right edge of the 80 column screen. One mo must be added and that's what bit 5 in the high byte is used for.

The GEOS and Wheels kernal routines, such as Rectangle, call another ro "NormalizeX" for adjusting the actual values of the registers prior to them. NormalizeX will take the doubling bits that we added to r3 and r4 make the values in r3 and r4 end up exactly as they should be.

PutString also calls NormalizeX to adjust the value in r11. However, I use the doubling bits on r11 as you can see in my routine AdjR11. Inste actually perform the same operation as what NormalizeX would do. Why? Because I can also use the same AdjR11 routine prior to calling PutChar speed reasons, PutChar doesn't call NormalizeX. It requires the actual coordinate be set up in r11. If I'm only going to be using PutString in program and never use PutChar or SmallPutChar, I can use doubling bits r11, because PutString calls NormalizeX.

So, what do we use AdjCards for? This is for bitmaps and icons in GEOS Wheels and color coordinates in Wheels as well as other routines that u byte coordinates for the horizontal position and width.

Let's say we want to place a bitmapped image on the screen.

```
  LoadB r1L,#20 ;left byte coordinate.
  LoadB r1H,#50 ;top pixel coordinate.
  LoadB r2L,#IMAGEWIDTH ;widthh in bytes.
  LoadB r2H,#IMAGEHEIGHT ;height in pixels.
  LoadW r0,#image ;point to the image.
  jsr AdjCards
  jsr BitmapUp ;display the bitmap.

image:

  ************ ;(this is supposed to be
  ************ ; a photo scrap)
  ************
  ************

IMAGEWIDTH==picW IMAGEHEIGHT==picH
```

Note: geoAssembler has a bug that requires you to always place a space and below the photo scrap image in your source code.

You can also use AdjCards prior to calling a dialog box if you have use DBUSRICON definitions. The only thing that needs adjusting is one value the icon table that your DBUSRICON definition points to.

Here's a dialogue box example:

```
sampleBox:
  .byte DEF_DB_POS

  .byte DBUSRICON,2,46
  .word sampleTable
```

```
  .byte OK,DBI_X_2,DBI_Y_2

  .byte 0
```

This DB will put a programmer defined icon two bytes from the left of t and 46 pixels from the top of it. An OK icon will also appear at the lo right corner.

Now, prior to calling DoDlgBox, we must modify the table that is at "sampleTable".

```
  ...
  lda sampleTable+4
  ora dblB
  sta sampleTable+4
  LoadW r0,#sampleBox
  jsr DoDlgBox
  ...

sampleTable:
  .word samplePic ;pointer to our icon.
  .byte 0,0,2,10 ;x coord at 2 bytes, y at 10 pixels.
  .word SampleRoutine ;routine to call when user clicks
                      ;on icon.
```

The above routine will set bit 7 of the 5th byte in the above table. Th value is 2 and it will become $82 when done. The next time this dialogu is used, it will still be $82. The kernal doesn't double the value in o own table. The value is transferred to the same registers used for displaying a bitmap and doubled there. So, we can safely keep on settin 7 in the table without causing the value to be doubled a second or thir fourth time, etc.

PAY SPECIAL ATTENTION TO THIS PART...

OK, so what about the other coordinates in the dialogue box above? What about that "OK" icon?

Don't worry about it. The kernal will take care of it for you. In GEOS and Wheels 128, the kernal looks at the very first byte in the dialogue In this case, we used DEF_DB_POS. When this is used, the kernal will automatically double any system icons and text locations if in 80 colum mode. But it won't double the values in a user defined icon because the program might be intended ONLY for 80 column use and might already have correct locations defined. So, it's up to the programmer to add the dou bits or double the actual values, like we already did in our sample.

Now for the important part. Sometimes we don't want to use the standard dialogue box or have it appear in the standard location. Instead of us DEF_DB_POS, we can use SET_DB_POS. But the next six bytes in the dialog box must define the location of the box on the screen. These are normal rectangular coordinates just like Rectangle would use. And if you set t correct bits, your dialogue box will appear in the same relative positi the 80 column screen.

PLUS... the kernal will look at the high byte of the left hand location determine if it should double any values within the dialogue box.

Here's a similar box with just an OK icon in it for this example:

```
anotherBox:
  .byte SET_DB_POS
  .byte 40 ;top of DB
  .byte 160 ;bottom of DB
  .word 80 ;left of DB
  .word 239 ;right of DB

  .byte OK,23,136 ;this will appear near the lower right.

  .byte 0 ;end of DB
```

Now, all we need to do is set up the doubling bits in the left and righ coordinates of the box. We can do it and then run the dialogue box:

```
  ...
  lda anotherBox+4
  ora dblB
  sta anotherBox+4
  lda anotherBox+6
  ora add1W
  sta anotherBox+6
  LoadW r0,#anotherBox
  jsr DoDlgBox
  ...
```

Pretty simple.

I used to have a routine at the very start of my applications that woul through and adjust all the coordinates thoughout the whole program. But wasn't easy. Everytime I made a change to one routine, I had to make su added the change to this modifying routine. It was a hassle. Plus, the routine can't adjust any values in a VLIR record until the record is lo into memory. That meant calling a different routine to adjust all the locations in the record when it was loaded. Each and every time it was loaded!

Now, I just do like I've described here. Each individual routine takes of itself. It's much easier this way. And as you can see, it's not real that tough to implement. Plus, I can copy a routine to another program without a lot of extra work.

The fact that all the routines share common registers makes it nice and easy. The routines that work with pixel coordinates use r2L, r2H, r3, a while the routines that deal with byte coordinates use r1L,r1H,r2L, and They are always used as follows:

pixel routines: r2L is used for top pixel. r2H is used for bottom pixel is used for left pixel. r4 is used for right pixel.

byte (or card) routines: r1L is used for left card. r1H is used for top pixel (or card in color routines). r2L is used for width in cards. r2H used for height in pixels (or cards in color routines).

Hope everybody can make use of these and write new software for the 40 column screen that can also be used on the 80 column screen.

-Maurice

Top of Page Message 49035 Previous
From: CS (CINDYSIMMS) To: Bo Zimmerman (BOZAC) UNREAD 6 of 6
Posted: 1/2/99 6:01 PM Reply to: 49020
Glad to see you are still programming. I use geoBeap to go to and from files quite a bit. Since GUS got killed, I have been looking into (yuck and Mac emulators of the C64 and geos. geoBEAP in 80 col on the 128 wit Wheels and SCPU support would speed up the process a bunch. Keep up the great work.

Reply Forward Delete

[All Messages] <<< 1-2 3-6

Bo Zimmerman

From: ARCA93@delphi.com
Sent: Wednesday, May 05, 1999 8:28 PM
To: gtm@videocam.net.au
Subject: Third email exchange with Roy...

Hi Maurice,

> You can load r1L,r1H with the track, sector of the desired subdirectory
> and call OpenDirectory. There is also OpenRoot. These routines exist
> within every Wheels driver, even the 1541. So it is safe to call any
> routine within any Wheels driver without causing an error. The drive
> will deal with it. For instance, no matter what r1L,r1H equals, the
> 1541 will always open up to track 18, sector 0.

Does the driver for a 1541 also call OpenDisk, when I call
OpenRoot/-Directory?

> TopDirectory
> UpDirectory
> DownDirectory

What do this routines do? Where are they located?

> It was never intended to work. This is because there is no such
> thing as StashBRAM or FetchBRAM, only MoveBData. MoveBData takes
> care of both stash and fetch by using a source bank and a destination
> bank. The source and destination can be the same, too. Likewise,
> only %00, %10, and %11 will work with DoBOp. Instead of using %01,
> you're supposed to use %00 and set the source and destination bank.

:-(In the book about the German "MegaAssembler" there I read, that %01
could be used. I tried this, because I needed 2 routines: One for copying
from bank 0 to bank 1 and another to copy from bank 1 to bank 0. I wrote
one routine for this 2 things and called this either with y=%00 or with
y=%01. After many system crashs I found the error in the routine DBOp
itself and I had to changed my program :-(

> To detect Wheels, version will contain $41 or higher. And also

Why $41?

Thank you for all the information. I think, I'll start to change my
DoubleDesk in the next days...

Bye
  Roy

==========

Hi Roy,

<<<
Does the driver for a 1541 also call OpenDisk, when I call
OpenRoot/-Directory?
>>>

Yes, OpenDisk gets called by OpenRoot and OpenDirectory
in all the drivers. The 1541/71/81 drivers just go straight
to OpenDisk. First they alter r1L and r1H to the correct
values for the directory header.

<<<

Bo Zimmerman

From: ARCA93@delphi.com
Sent: Wednesday, May 05, 1999 8:28 PM
To: gtm@videocam.net.au
Subject: Third email exchange with Roy...

Hi Maurice,

> You can load r1L,r1H with the track, sector of the desired subdirectory
> and call OpenDirectory. There is also OpenRoot. These routines exist
> within every Wheels driver, even the 1541. So it is safe to call any
> routine within any Wheels driver without causing an error. The drive
> will deal with it. For instance, no matter what r1L,r1H equals, the
> 1541 will always open up to track 18, sector 0.

Does the driver for a 1541 also call OpenDisk, when I call
OpenRoot/-Directory?

> TopDirectory
> UpDirectory
> DownDirectory

What do this routines do? Where are they located?

> It was never intended to work. This is because there is no such
> thing as StashBRAM or FetchBRAM, only MoveBData. MoveBData takes
> care of both stash and fetch by using a source bank and a destination
> bank. The source and destination can be the same, too. Likewise,
> only %00, %10, and %11 will work with DoBOp. Instead of using %01,
> you're supposed to use %00 and set the source and destination bank.

:-(In the book about the German "MegaAssembler" there I read, that %01
could be used. I tried this, because I needed 2 routines: One for copying
from bank 0 to bank 1 and another to copy from bank 1 to bank 0. I wrote
one routine for this 2 things and called this either with y=%00 or with
y=%01. After many system crashs I found the error in the routine DBOp
itself and I had to changed my program :-(

> To detect Wheels, version will contain $41 or higher. And also

Why $41?

Thank you for all the information. I think, I'll start to change my
DoubleDesk in the next days...

Bye
Roy

==========

Hi Roy,

<<<
Does the driver for a 1541 also call OpenDisk, when I call
OpenRoot/-Directory?
>>>

Yes, OpenDisk gets called by OpenRoot and OpenDirectory
in all the drivers. The 1541/71/81 drivers just go straight
to OpenDisk. First they alter r1L and r1H to the correct
values for the directory header.

<<<

> TopDirectory
> UpDirectory
> DownDirectory

What do this routines do? Where are they located?
>>>

TopDirectory takes you to the root and can safely be called with
any driver. UpDirectory takes you to the parent directory of the
subdirectory you're currently in. DownDirectory will open a
subdirectory that's in the current directory.

```
;this takes you to the root directory.
GoToRoot:
  lda #(5|64)
  jsr GetNewKernal
  jsr TopDirectory
  jmp RstrKernal

;this takes you to the parent directory.
GoToParent:
  lda #(5|64)
  jsr GetNewKernal
  jsr UpDirectory
  jmp RstrKernal

;this opens a subdirectory.
;dirEntryBuf must be loaded with a subdir's directory entry
;prior to calling this routine.
GoToSubdir:
  lda #(5|64)
  jsr GetNewKernal
  jsr DownDirectory
  jmp RstrKernal

GetNewKernal==$9d80
RstrKernal==$9d83
TopDirectory==$500f
UpDirectory==$5012
DownDirectory==$5015
```

<<<
:-(In the book about the German "MegaAssembler" there I read, that %01
could be used. I tried this, because I needed 2 routines: One for copying
from bank 0 to bank 1 and another to copy from bank 1 to bank 0. I wrote
one routine for this 2 things and called this either with y=%00 or with
y=%01. After many system crashs I found the error in the routine DBOp
itself and I had to changed my program :-(
>>>

The Hitchhiker's Guide to GEOS is also incorrect in the description
of DoBOp.

<<<
> To detect Wheels, version will contain $41 or higher. And also
Why $41?
>>>

When I first started doing this project, it was to be called GEOS 3.0.
Geoworks said it couldn't be called that because GEOS in the Brother
GeoBook was called 3.0. So, I figured I'd make it sound even better.
Wheels 64 started out as V4.0. The new upgrade is V4.2, but the
kernal version number is still 4.1. Mostly the 4.2 upgrade over
4.1 consists of the supporting applications such as the Dashboard
being changed. The V4.1 kernal has significant differences over the

4.0 kernal. I'm not supporting the 4.0 kernal any more. Every registered owner of Wheels is getting the latest upgrade for free.

When you install a default desktop in Wheels, you'll want to follow the same method that I've designed for this purpose. This allows multiple desktops to be used. Upon exiting a desktop, the one that called it will be returned to. As many as 16 desktops can be chained together. For instance, you can be working in the Dashboard and then load in geoSHELL. geoSHELL can then load up Concept, my assembler/linker program. Since Concept installs as the default desktop, it can launch geoWrite and you'll return to Concept when you exit geoWrite. If you exit Concept, you'll return to geoSHELL. And geoSHELL can exit back to the Dashboard.

```
numDesktops==$88a6
dtDrive==$8868
dtPartition==$8869
dtType==$886a


MakeDefault:
  lda numDesktops     ;max of 16 desktops already installed?
  cmp #16
  bcs 20$             ;branch if so.
  LoadW r0,#$c3cf     ;point to 64 desktop name.
  bit c128Flag        ;or are we running on the 128?
  bpl 5$              ;branch if 64.
  LoadW r0,#$ca01     ;point to the 128 desktop name.
 5$
  LoadW r1,#progName  ;point to our own desktop's name.
  ldx #r0
  ldy #r1
  jsr CmpString       ;are we already the default desktop?
  beq 20$             ;branch if so. Already installed.
  LoadW r1,#deskName  ;start building an entry for
  LoadW r2,#13        ;the desktop that is calling us.
  jsr MoveData
  MoveB dtDrive,deskName+13 ;save the calling desktop's drive.
  MoveB dtpartition,deskName+14 ;and its partition.
  MoveB dtType,deskName+15 ;and its drive type.
  LoadW r0,#deskName
  jsr StashDTName     ;now stash this info into the REU.
  LoadW r0,#progName  ;now we install ourself as the
  lda #10             ;new default desktop.
  jsr GetNewKernal    ;the OS will do it for us.
  MoveB curDrive,dtDrive ;make this the current desktop drive.
  jsr GetHeadTS       ;find out what partition we're in.
  MoveB r2L,dtPartition ;and make it the desktop partition.
  MoveB curType,dtType ;and make this the desktop drive type.
 20$
  rts

progName:
  .byte "YourName",0

;call this when the user wishes to quit the current desktop
;and return to the calling desktop.
ExitToDesktop:
  LoadW r0,#deskName  ;get the calling desktop's name
  jsr FetchDTName     ;and drive info.
  MoveB deskName+13,dtDrive
  MoveB deskName+14,dtPartition
  MoveB deskName+15,dtType
  lda #10             ;let the OS reinstall the
  jsr GetNewKernal    ;calling desktop.
  jmp EnterDesktop
```

```
StashDTName:
  ldy #STASH          ;stash the calling desktop's
  jsr MoveDTName      ;name and info into the REU.
  inc numDesktops     ;increment the number of desktops.
  rts

FetchDTName:
  dec numDesktops     ;decrement the number of desktops
  ldy #FETCH          ;and fetch the calling desktop's
  jsr MoveDTName      ;name and other info.
;fall through...
MoveDTName:
  LoadB r1H,#]$fe00   ;point to within the $fe00 page
  lda numDesktops
  asl a
  asl a
  asl a
  asl a
  sta r1L
  LoadW r2,#16
  LoadB r3L,#0        ;bank 0 in the REU.
  jmp DoRAMOp

deskName:
  .block 16
```

-Maurice

Bo Zimmerman

From: ARCA93@delphi.com
Sent: Wednesday, May 05, 1999 8:27 PM
To: gtm@videocam.net.au
Subject: Second email exchange with Roy...

Hi Maurice,

> curType

Oh, it's good, that now the FD hasn't the same data as the HD. By the way, my brother's cd-rom driver is indicated by $41, when a D64-file is opend and by $45 when a normal DOS-directory is opened.
What about the RL as $83? How can a get it's partition list, when I don't have its unit-number (when it's indicated as $33, I can use OPEN..."$").
Have I to read the partition table by normal RL-Routines ($DExx)?
Another question: How can I change the native subdirectories? Are there the same routines like under GateWay $9050 for changing to root and $9053 for changing to the subdirectory indicated by r1L&H)?

Where is the routine GetHeadTS indicated?

In German GEOS there is an error in the menu routine: No menu can be have a right margin greater than 255. In my DoubleDesk I included a patch for this. Do you know this error or isn't it there is US GEOS?
There are some more error, which I found: In the original ColorRectangle-routine, NormalizeX isn't called. The DoBOp-routine doesn't work, when .y is loaded by %01 (only with %00 %10 and %11).

What about version? Can I find $20 or $30 there? What can I find in bootName (under GEOS it's "GEOS BOOT" and under GateWay "GATEWAY")?

Because no switcher is included in Wheels, it would be interesting to know, if geoHexer works under your system.

Must I allocate a used RAM bank under Wheels 128, because for desk accesories C128-bank 1 is used?

Bye
  Roy

==========

Hi Roy (and Ronny),

<<<
What about the RL as $83? How can a get it's partition list, when I don't have its unit-number (when it's indicated as $33, I can use OPEN..."$").
>>>

The following routine will return the device number of the RamLink in .x:

```
GetRLNumber:
  lda #(5|64)
  jsr GetNewKernal
  jsr FindRamLink
  jmp RstrKernal
```

If x=0 after calling the above routine, then no RamLink found. Otherwise, x will hold the REAL device number of the RamLink no matter what kind of driver is being used to control it.

<<<
Have I to read the partition table by normal RL-Routines ($DExx)?
>>>

Once you've got the device number of the RL, you can read the partition table as you normally would. The best way is to just let the OS display the partition requestor to the user and upon return, the current partition will be the one selected by the user. This keeps code in your application to a minimum.

The following will put up a file requestor allowing the user to select a partition or even a subdirectory within the partition.

```
SelectPartition:
  lda #(5|64)
  jsr GetNewKernal
  jsr ChPartition
  jmp RstrKernal
```

Upon return, you can call GetHeadTS to find out the current partition number in r2L and the track and sector of the directory header in r1L,r1H.

GetHeadTS==$9063

<<<
Another question: How can I change the native subdirectories? Are there the same routines like under GateWay $9050 for changing to root and $9053 for changing to the subdirectory indicated by r1L&H)?
>>>

You can load r1L,r1H with the track, sector of the desired subdirectory and call OpenDirectory. There is also OpenRoot. These routines exist within every Wheels driver, even the 1541. So it is safe to call any routine within any Wheels driver without causing an error. The drive will deal with it. For instance, no matter what r1L,r1H equals, the 1541 will always open up to track 18, sector 0.

OpenRoot==$9050
OpenDirectory==$9053

This is just like in gateWay. This remains the same for compatibility reasons. The disadvantage to this is that the DOS in the drive is still looking at the previous directory in case you do any direct DOS calls. For this there are the following routines:

TopDirectory
UpDirectory
DownDirectory

They each update the tables within the ram of the drive, including the RamLink if the call is made for the RamLink.

<<<
In German GEOS there is an error in the menu routine: No menu can be have a right margin greater than 255.
>>>

This error is fixed in Wheels. You can have menus all the way to the right of the screen now.

<<<
Do you know this error or isn't it there is US GEOS?
>>>

Yes, the error also existed in GEOS in the US, but not in Wheels.

<<<
There are some more error, which I found: In the original ColorRectangle-routine, NormalizeX isn't called.
>>>

I think the reason NormalizeX wasn't called was because it was intended that ColorRectangle would be called after Rectangle was called. NormalizeX was already called in Rectangle. Anyway, I left that part the same but incorporated some new color routines that work better.

The old ColorRectangle is renamed to ColorBox. The new ColorRectangle works differently. Instead of supplying pixel coordinates, you supply card coordinates. There is also a routine called ConvToCards that will convert pixel coordinates to card coordinates for you. The card coordinates are done very similar to the way they are done when displaying a photo scrap. Left card, top card, width and height. The current color mode is accomodated. It works with 8x8, 8x4, and 8x2. It also works with 16K VDC ram and will simply ignore doing color if color mode 0 is being used.
  You can also use 40 column color values and a suitable 80 column color will be substituted. Or you can use actual VDC color values. You can set bit 7 in the left card byte and also in the width byte and it will be normalized automatically.

<<<
The DoBOp-routine doesn't work, when .y is loaded by %01 (only with %00 %10 and %11).
>>>

It was never intended to work. This is because there is no such thing as StashBRAM or FetchBRAM, only MoveBData. MoveBData takes care of both stash and fetch by using a source bank and a destination bank. The source and destination can be the same, too. Likewise, only %00, %10, and %11 will work with DoBOp. Instead of using %01, you're supposed to use %00 and set the source and destination bank.

<<<

What about version? Can I find $20 or $30 there? What can I find in bootName (under GEOS it's "GEOS BOOT" and under GateWay "GATEWAY")?
>>>

You'll find "GEOS BOOT" there.
To detect Wheels, version will contain $41 or higher. And also another location should be checked, driverVersion. It should be $51 or higher.

driverVersion==$904f

All GEOS drivers and gateWay drivers will have values less than $50 in this location.

<<<
Must I allocate a used RAM bank under Wheels 128, because for desk accesories C128-bank 1 is used?
>>>

You mean bank 0? In Wheels 128, desk accessory code is swapped with bank 0 ram in the 128. Therefore, you have complete free use of the first 32K of ram in the REU's bank 0. It's already allocated for the application to use. When your application exits, then the Dashboard will use it.

A task switcher would be more difficult to deal with because it would also have to save this 32K of REU space. Plus it would have to save some OS code that resides in the REU and also the Dashboard's REU space in the last bank. Too much work. I'm saving multitasking for the SCPU version of Wheels.

-Maurice

Bo Zimmerman

From: ARCA93@delphi.com
Sent: Wednesday, May 05, 1999 8:26 PM
To: gtm@videocam.net.au
Subject: Email exchange with Roy Bachmann

Here's some edited emails exchanges that I've had during the last few days with Roy Bachmann of Germany...

==========
Hello Maurice,

some days ago I had seen your Wheels at a C64/128 club meeting. It's fantastic, exspecially the managing of the disk drivers. Now I'm working with the GateWay-System, because I only work with native RL/FD-partitions. Also my desktop "DoubleDesk 128" works only with the GateWay because of this. But I heard of many SuperCPU users (I have no one), that there are many problems with GateWay and SCPU. So the can't use the DoubleDesk :-(Now I had the idea to buy Wheels and make my DoubleDesk compatible to it.

Now me question:
Is there a manual, where the differences between GEOS and Wheels can be read?

Exspecially interesting for me is:
Where can I find information, which parts of a REU are used and which are free?
Is the data in curType (or driveTypes) the same like under GateWay?
What happens, when I switch for example between a 1581 and a native partition?

Is the RamLink used as a normal drive like under GateWay (as unit 8 till 11) or as a RAM drive (as unit 12 or higher)?
Are there new routines in the disk drivers (for example for changing the partition)?
What other new routines are included (perhaps a better ColorRectangle-routine)?

And, of course: When can we get a special German version of Wheels?

Bye
Roy Bachmann

==========
And now my response to the above email...
==========

Hi Roy,

<<<
Is there a manual, where the differences between GEOS and Wheels can be read?
>>>

I haven't created a programming manual yet, but I plan to. So far, I've always answered any questions any programmer has had and also provided any information I could. I'm going to set up a section on my web site and start posting programming information there.

<<<
Exspecially interesting for me is:
Where can I find information, which parts of a REU are used and which are free?
>>>

For the currently running application, the first 32K of ram bank 0 is available. If running in Wheels 64, this is also used for a swap space whenever a desk accessory is loaded. If the app doesn't allow DAs, then this is not a problem. But if it does, then it will have to reload whatever code it stores in the REU.

When the current desktop is loaded, it can also use this area just like any application can. The current desktop can also use $4000-$b1ff in the system bank of the REU. This is the last bank of the REU and is normally not accessible to applications. The routine such as StashRAM and FetchRAM won't allow access to this area. But by incrementing the ramExpSize variable, they allow access. Then when finished, the variable should be decremented.

<<<
Is the data in curType (or driveTypes) the same like under GateWay?
>>>

The upper nybble of the drive type indicates the device while the lower nybble indicates the format type.

```
TYPE_CBM=$00
TYPE_FD=$10
TYPE_HD=$20
TYPE_RL=$30
TYPE_RAM=$80

DRV_1541=$01
DRV_1571=$02
DRV_1581=$03
DRV_NTV=$04
```

Now, if you combine the above values, you'll have the following:

1541: $01
1571: $02
1581: $03
1581 using FD native disk: $04
FD w/1581 disk or partition: $13
FD w/native partition: $14
HD w/1581 disk or partition: $23
HD w/native partition: $24
RL/RD w/1581 partition: $33
RL/RD w/native partition: $34
RAM1541: $81
RAM1571: $82
RAM1581: $83
RAMNATIVE: $84

A RamLink can also be type $83 if the device number is 12 or higher and if running in a 1581 partition. This is for compatibility with the way it was done in GEOS. If the RamLink is 8-11, then it's treated similar to the way gateWay handled it and it would be $33 or $34. This allows 1581 and native partition to be used.

There is a variable in each driver called "cableType" at $9073. With the HD, if bit 7 is set, the parallel cable is being used. With the RAM1581 drivers, if bit 7 is set, then this is a RamLink. If cleared, then it's a normal ramdisk running in an REU.

<<<
What happens, when I switch for example between a 1581 and a native partition?
>>>

If you use the OS routines for switching partitions, the correct driver will be installed for you. Switching partitions is easy.

```
;call this with .x holding the desired partition number.
GotoPartition:
  lda #(5|64)
  jsr GetNewKernal
  jsr GoPartition
  jmp RstrKernal
```

That routine above will switch to whatever partition number is in .x when you call it. The routine GetNewKernal calls in a portion of the Wheels kernal that is in the REU. RstrKernal restores the memory where the kernal was brought into. The routine GoPartition takes care of all the work for you. It will get the correct driver and install it and will switch partitions for you. If any error occured, you can check .x after calling the above routine.

There are also routines to call to display the dialogue box for the user to pick his own partition. And you can easily determine the current partition you're in. You do this with the routine GetHeadTS. This routine is contained in every disk driver. It will return the current t,s of the directory header in r1L and r1H. And r2L will hold the current partition number. If you check this when your program first loads, you can always know where to find your application if you need to access a module from it. This routine even works with the 1541. It will always return #1 in r2L so it's safe to call this with any device. It's also safe to attempt to change partitions on any device even if the device doesn't support it. This way, you don't have to check the device first. It saves a lot of code in your application.

<<<
What other new routines are included (perhaps a better
ColorRectangle-routine)?
>>>

Wheels 64 has a new ColorRectangle routine and it's different
from the original one in GEOS 128. Plus Wheels 128 has the
same routine as Wheels 64 and it also has the original one
that GEOS 128 had, only it's slightly improved.
Also, you can call the color routines on the 16K 128's without
worrying about crashing the machine. The color routines will
simply return without doing anything if the current screen
mode doesn't support it.

-Maurice

Here's some more stuff that I had from another note to some
other programmers:

==========

Wheels 64 and 128 programmers,

Here's some prelimary (and brief) info on how to allocate ram
from the REU device for use by a Wheels application.

First, some equates and symbol definitions:

```
GetNewKernal==$9d80
RstrKernal==$9d83

GetRAMBam==$5000
PutRAMBam==$5003
AllocAllRAM==$5006
AllocRAMBlock==$5009
FreeRAMBlock==$500c
GetRAMInfo==$500f
RamBlkAlloc==$5012
RemoveDrive==$5015
SvRamDevice==$5018
DelRamDevice==$501b
RamDevInfo==$501e
```

The ram handling routines reside within the "extended kernal"
which is stored in the last bank of the REU. Only a small
portion of the extended kernal is used for the ram handler.
There is much more in the extended kernal than this. But for
this discussion, we will talk only about the ram routines
and how to access them.

To bring the extended kernal into memory requires calling
the routine "GetNewKernal". But you must also indicate which
portion of the extended kernal you wish to access. The ram
routines are contained in what is known as Group 0. You load
the accumulator with a 0 in the lower nybble and also set
bit 6 and then call GetNewKernal as follows:

```
  lda #(0|64)
  jsr GetNewKernal
```

What this does, is it will swap the memory at $5000-$5fff with
that of the area of the REU that holds the ram handling routines.
By setting bit 6, you are telling GetNewKernal to swap the
memory but don't run any of the code. If bit 6 was clear, then
the first routine in this group would be executed and upon
termination, the memory would be swapped back. But we want to

use more than the first routine, so we'll want to leave the code in CPU memory until we're done with it.

Now, that the ram handler is in memory, we want to find out if there is any ram available for use. Ram from the REU device must be allocated in chunks of 64K at a time. Even if your program only needs 1 byte, you have to allocate a whole 64K bank. Let's see how we can find out how much ram is available.

```
jsr GetRAMInfo
```

It's as simple as that. Now, all you have to do is check a few registers.

r4H contains the total number of free 64K banks.
r2L contains the total number of consecutive free 64K banks.
r3L contains the starting bank of the largest free area.

At this point, you can choose to allocate the ram or just go ahead and use it. Be aware that if you don't allocate it and your program uses a desk accessory that can also allocate ram, it might step all over your ram bank. But if you're just going to use the ram and be done with it and then exit, don't worry about allocating it, because you will have to also deallocate it upon exit. Or the ram won't be available for use by other applications.

But in either case, you already know that r3L contains a bank number that is available for use, provided r2L is a non-zero number. If r2L=0, then no ram is available and you should inform the user with a message dialogue box.

Now, when it comes to allocating ram, you can allocate 1 or more banks at a time. You can choose to just allocate the ram and deallocate it upon exiting your application. Or you can add your ram bank(s) to the partition table. This provides an added safeguard in case the partition table is ever validated to free up allocated yet unused ram banks. Having your ram listed within the partition table is really only necessary if you wish to keep your ram banks allocated during a lengthy computing session, such as what the Toolbox does when it assigns ram banks to a ramdisk.

By placing your ram allocation in the partition table, you can always reuse the same banks each time the user runs your application. This is purely a matter of choice and also depends on the design of your application and whether or not you will need to allocate ram and/or store its listing in the partition table.

If you only want to allocate ram, load r2L with the number of 64K banks needed. Load r3L with the desired starting bank or 0 if you wish to let the OS assign a starting bank. And then call RamBlkAlloc.

```
LoadB r2L,#1  ;we need one bank of ram.
LoadB r3L,#0  ;let the OS pick a bank for us.
jsr RamBlkAlloc
```

If the ram allocation was successful, .x will equal 0, otherwise .x will equal INSUFF_SPACE. If .x =0 then r3L will be holding the bank that was just allocated.

Now, this same bank can be used with the normal StashRAM, FetchRAM, and SwapRAM routines which are in the standard kernal. Now that we've allocated our ram, let's put the

extended kernal back and continue. This is simple:

```
jsr RstrKernal
```

So, here it is again for an application that just needs to allocate one bank of ram:

```
Get1Bank:
  lda #(0|64)      ;swap group 0
  jsr GetNewKernal ;into memory.
  LoadB r2L,#1     ;we want one bank.
  LoadB r3L,#0     ;and don't care where it begins.
  jsr RamBlkAlloc  ;get a bank of ram.
  jmp RstrKernal   ;.x is preserved.
```

Put the above routine in your source code and call it when you need to allocate a bank of ram for your own use, like so:

```
  jsr Get1Bank
  txa
  bne 90$
  MoveB r3L,ourOwnBank
  rts
 90$
  ... display an error message here

ourOwnBank:
  .block 1
```

See how easy it is?

Now, how do we free up this bank of ram when we're ready to quit our application?

```
FreeOurBank:
  lda #(0|64)
  jsr GetNewKernal     ;get group 0 into memory.
  jsr GetRAMBam        ;get the ram bam into the workspace.
  MoveB ourOwnBank,r6L ;indicate which bank to free up.
  jsr FreeRAMBlock     ;and then free it up.
  jsr PutRAMBam        ;write the modified ram bam.
  jmp RstrKernal       ;and restore the kernal and memory.
```

As you can see, it's a little different to free up one bank at a time. If we had listed our ram in a partition table, we could have sent a request to free up all the ram banks listed in our partition, but this example isn't doing that, so we must do it as in the above routine.

One thing you must know. The code you're using to call the extended kernal can't reside within $5000-$5fff, because your code will get swapped out when the extended kernal is brought in.

That's all there is to it when we need a bank of ram. Now how do we access the bank once we've got one to use? Simple, we just use the same routines that have always been available since GEOS 2.0. These routines are in the standard kernal and don't need anything special to access them. In fact, you can call them from almost any memory configuration. It doesn't matter if you've already called InitForIO or not, you always have access to these ram access routines. They are as follows:

```
StashRAM==$c2c8
FetchRAM==$c2cb
```

```
SwapRAM==$c2ce
VerifyRAM==$c2d1
DoRAMOp==$c2d4
```

It's easy to figure out what the above routines do by their names, except for the last one. However, DoRAMOp is the actual workhorse of the whole bunch. Each of the first four routines will call DoRAMOp to do the actual work.

In each case, you will load the following registers:

r0 holds the CPU address.
r1 holds the REU address.
r2 holds the number of bytes
r3L holds the REU bank.

So, if we want to stash 50 bytes of ram located at $2000 to the beginning of our bank of ram in the REU, the following will get the job done:

```
  LoadW r0,#$2000
  LoadW r1,#$0000
  LoadW r2,#50
  MoveB ourRamBank,r3L
  jsr StashRAM
```

Not too hard, huh?

If you want to bring those same 50 bytes back into the computer, just use FetchRAM instead of StashRAM. If you want to swap the 50 bytes of computer memory with the 50 bytes of REU memory, just use SwapRAM.

So, what does DoRAMOp do? Actually, StashRAM, FetchRAM, SwapRAM, and VerifyRAM only load .y with a value. DoRAMOp performs the desired job according to the value in .y. So, you could also do the same jobs by loading .y with a value and calling DoRAMOp to do the job. Check this out:

```
STASH=$90
FETCH=$91

StashBytes:
  ldy #STASH
  .byte 44
FetchBytes:
  ldy #FETCH
  LoadW r0,#$2000
  LoadW r1,#$0000
  LoadW r2,#50
  MoveB ourRamBank,r3L
  jmp DoRAMOp
```

See how you can make your code shorter by combining two routines? Of course this example is only good if you're always moving the same bytes to/from the same locations.

An added benefit of using these system routines is that you, as a programmer, don't even need to deal with the type of REU being used. All these routines work just as well with a 1750, geoRAM, RamLink DACC, and the SuperRAM. Any ram device used in Wheels can use these routines.

Before long, I hope to have available "A Thumber's Guide to Wheels Programming".

Until then, all you have to do is ask for the info and I will respond as I did here.

-Maurice

p.s. See how easy it would be to patch up geoCanvas for Wheels?

Bo Zimmerman

From: ARCA93@delphi.com
Sent: Sunday, May 23, 1999 8:01 PM
To: gtm@videocam.net.au
Subject: REU banks, and installing drivers.

Hi Roy,

<<<
Here are more questions about Wheels:

Is the option to use the REU for MoveData always inactive under Wheels?
>>>

Yes, MoveData is gone. I felt it wasn't all that important anymore, so I eliminated it. That helped to free up some code space in the kernal.

<<<
Is the printer driver still located at $d8c0/$d9c0 under Wheels 128?

What have I to do to write an install routine for your new input drivers under Wheels 128?
>>>

Yes, the printer driver header is still at $d8c0 in bank 1 and the driver code is at $d9c0 in bank 1, just like in GEOS 128. However, you shouldn't manually place the driver there unless you absolutely have to. There is a kernal routine for installing input and printer drivers now. The same routine takes care of both. The following code will install a printer or input driver:

```
InstallDriver==$5006

InstIODriver:
  lda #(10|64)
  jsr GetNewKernal
  jsr InstallDriver
  jmp RstrKernal
```

Simply call the above routine with dirEntryBuf holding the directory entry of the driver you wish to install. InstallDriver will check to see if the file is a printer driver or an input driver and act accordingly. This works for both Wheels 64 and Wheels 128. Very Simple.

<<<
I've written a printer driver for ESC/P2 printers which is so long that it uses an own RAM bank. There is an install program for this driver, which looks for an empty RAM bank (what is very easy under Wheels :-). It allocates this bank, stores some data into it. Then it stores the number of this bank into the kernal at $bfff (if there are problems, because another program uses this adress, the user can choose another adress) and so the printer driver itself can load parts of this data from this bank. But there's a problem under Wheels: The install program allocates a bank and when you leave GEOS and reboot then, this bank is still allocated. So the install program looks for a new bank and allocates it. So after some reboots the REU is full and I must open the Toolbox to free these banks. Have you an idea what to do?
>>>

What you should do is use the kernal routines in Wheels to allocate the bank, and assign a partition name to it. That way, you don't have to store any information anywhere in the kernal. You can just check to see

which bank is assigned to the partition you have. If your partition isn't there, then you know you have to allocate a new bank. Use the following routine to allocate a 1 bank partition:

```
SvRamDevice==$5018

GetBank:
  lda #(0|64)
  jsr GetNewKernal
  LoadB r2L,#1      ;1 64K bank needed.
  LoadB r7L,#72     ;ID number, can be anything less than 128.
                    ;(only ramdisks can have 128 and higher ID numbers)
  LoadW r0,#partName ;point to a partition name.
  ldy #0            ;don't care which partition...
  sty r3L           ;or bank to use.
  jsr SvRamDevice   ;allocate a bank and partition.
  jsr RstrKernal
  txa               ;was a bank and partition available?
  bne 90$           ;branch if not.
  MoveB r3L,ramExpBank  ;this is the bank number.
  sty reuPartNumber ;and this is the partition number.
 90$
  txa
  rts

partName:
  .byte "Bachmann",0  ;use whatever partition name you want.
ramExpBank:
  .block 1
reuPartNumber:
  .block 1
```

The routine you use to call the above routine can test the equals flag to see if it was successful.

Whenever your printer driver is first called, you can use the following routine to see if you already have a bank assigned:

```
RamDevInfo==$501e

IsBankAssigned:
  lda #(0|64)
  jsr GetNewKernal
  ldy #1
 10$
  jsr RamDevInfo
  lda r3L          ;is this partition in use?
  beq 70$          ;branch if not.
  lda r7L
  cmp #72          ;does this partition have our ID number?
  bne 70$          ;branch if not.
  LoadW r0,#partName
  ldx #r0
  ldy #r1
  jsr CmpString    ;check the partition name.
  bne 70$          ;branch if not "Bachmann"
  jsr RstrKernal
  lda r3L          ;exit holding the bank number.
  rts
 70$
  iny
  cpy #9           ;check all 8 partitions.
  bcc 10$
  jsr RstrKernal
  lda #0
  rts
```

When you call this routine, it will return the bank number in the accumulator. If zero, then you don't have a bank assigned yet. Also, y will be holding the partition number. r1 is no longer pointing to the partition name once RstrKernal is called.

When you're ready to free up your bank, just call the following routine:

```
DelRamDevice==$501b

FreeUpBank:
 jsr IsBankAssigned   ;do we have a bank assigned?
 beq 90$              ;branch if not.
 lda #(0|64)
 jsr GetNewKernal
 jsr DelRamDevice     ;y holds the partition number.
 jsr RstrKernal
90$
 rts
```

In the above routine, we loaded y with the partition number when we called IsBankAssigned. DelRamDevice only needs the partition number to delete the partition and free up the ram bank.

You can allocate banks without using a partition. But by using a partition, it's easy to identify your own ram and to find out where it is and how big it is. In your case, you're only using one bank. If you had more banks, you can find out how many because RamDevInfo returns the number of banks in r2L.

(By the way, $bfff is not a good idea to use.)

-Maurice