Concept and Concept+

The integrated assembler/linker package for Wheels 64 and Wheels 128.
Operates in both 40 and 80 column modes on the 128.

Assembler and linker operation is based on the original GEOASSEMBLER and GEOLINKER from the GeoProgrammer package, by Berkeley Softworks.

If you're familiar with using GeoProgrammer, then you'll easily understand how to use this software. If you do not already own GeoProgrammer, I highly recommend that you obtain your own copy of it. It may be downloaded free of charge from:

http://cmdrkey.com

Or it can be purchased for $20 which includes the complete printed manual and disks from Click Here Software Co.

1-866-CMDRKEY (order line)
1-517-543-5202 (for information)

Also available is the "Official GEOS Programmer's Reference Manual" and "The Hitchhiker's Guide To GEOS". Both of these manuals are essential for software development in GEOS and Wheels.

Wheels 64 and Wheels 128 may also be ordered from Click Here Software.

Concept

The integrated assembler/linker package for Wheels 64 and Wheels 128.
Operates in both 40 and 80 column modes on the 128.

Assembler and linker operation is based on the original GEOASSEMBLER and GEOLINKER from the GeoProgrammer package, by Berkeley Softworks.

This guide assumes you are already familiar with the use and functions of the geoProgrammer package which includes both geoAssembler and geoLinker.

USING CONCEPT

Concept may be used to create software for GEOS or Wheels as well as non-GEOS software that would run from BASIC. You can also create software for other 6502 based computers and/or other equipment. In addition, Concept+ can be used to create software for 65C02 and 65816 based machines.

You'll find that using Concept to do your programming will be just about the easiest assembler program you've ever used. Load and run it just like you would any other GEOS or Wheels application. When it first starts up, you're presented with a screen showing 6 icons. Each icon allows you to perform a specific task. In the top row of icons, one is for the assembler, one is for the linker, and one allows you to turn on or off the .dbg and .sym file generators. In the lower row of icons, one icon allows you to launch GeoWrite, another allows you to choose from a file requestor any application, desk accessory, or auto-exec file to run, and the last icon quits Concept and takes you back to the Dashboard.

When Concept is loaded into memory, it installs itself as your default desktop. This is how it is able to run applications such as GeoWrite. Upon exiting GeoWrite, you'll be returned right back to Concept. This makes it very easy to edit, assemble, and link your source code files. And since you can run any app, DA, or auto-exec, you can even test your newly created software right from Concept.

IMPORTANT: If you're using Concept in a native partition or native ramdisk, you must put it either in the root directory or within the system directory that's visible to the root directory. This is so that the operating system can find it when you exit an application. If it's buried within a subdirectory, you'll be requested to insert a disk containing Concept. Try to avoid this.

Concept loads itself entirely into memory. There is no need to keep a disk in the drive with Concept on it, except for when you exit an application to return to Concept.

When you launch the assembler or linker, you have the option of choosing a source directory and a destination directory. In the dialogue boxes for choosing the source and destination, there is also a "Subdir" icon. By clicking on this, you can change partitions or subdirectories. This allows you to read source code and assemble anywhere on your system. All four drives are available for use.

While assembling or linking, the source directory is always checked for the source files first. If a needed file is not found, then the currently open partitions on the other 3 drives will be searched also.

IMPROVEMENTS FROM GEOPROGRAMMER

Many of the bugs that you've grown used to in GeoProgrammer are now gone in Concept. You might not notice this right away or maybe never. But here's a few of the things that are gone for good:

Try this with geoAssembler:

```
    lda     r0L,y
```

You'll get an "Illegal addressing mode" error. The cure was always to do something like this:

```
    .byte   $b9,[r0L,]r0L
```

Concept recognizes what you want to do and instead of giving you an error, it assembles the code as absolute,y just like you intended.

Sometimes you may have troubles getting symbols properly recognized. GeoAssembler had a problem in the way it would resolve symbols. Sometimes it would flag them incorrectly as it passed the remainder of the task along to the linker. And other times, it would resolve a symbol when it didn't really know the value of the symbol yet. In those cases, only the linker can resolve a symbol. These problems are now gone with Concept.

GeoAssembler always wanted you to add an "a" to any accumulator opcode:

```
asl     a
```

This is not a big deal, but nevertheless, it's something you have to remember to do. Concept let's you put the "a" in there, or you can leave it out. It's up to you.

When you get errors during assembling or linking, you have the option of launching GeoWrite immediately to view the error file. Concept will launch the correct version of GeoWrite depending on if you're using Wheels 128 or Wheels 64. The .err files as well as the .sym files are formatted using BSW128. This font will show up in GeoWrite 64 as plain BSW. So, the same files will look good on either system.

One thing you might notice if you're paying attention is that each module listing in the .sym file ends with a formfeed. GeoLinker failed to do this and you could cause GeoWrite to crash if you tried to edit the text. Each page ended with a zero byte and GeoWrite doesn't appreciate that. A zero byte at the end of a GeoWrite page can play havoc with GeoWrite. Feel free to edit an .err file or .sym file that was created by Concept.

In the .err files, the page numbers and line numbers for the errors have been eliminated. This was done mainly because it's too difficult to keep track of the exact position within the source code where the error occured. When a source code file is read in, all but the ascii text is stripped and precise positioning is lost. This is evidenced by the fact that GeoProgrammer never showed the correct location of the errors. Concept does this differently now. It will display the name of the routine where the error occurs. Plus, in many cases, the actual source code line is displayed. Using GeoWrite's search function will take you right to the error in question.

GeoLinker would always go nuts if you had more than 99 errors in your files. Concept keeps on going through the rest of the linking process, but stops recording errors when it reaches 99. Take care of those 99 errors first and link again to see more.

Whenever you had too many local labels in a routine, GeoAssembler would fill up your .err file with many meaningless local label errors. Chances are, you had no idea where the actual error was. Concept shows one error, the routine where the excess

local labels are.

There were times when you might forget to put a local label in a routine, even though you had something in the routine that was making reference to one. And sometimes, GeoAssembler wouldn't catch this error. This really made for some tough debugging sessions. Concept catches all the local label errors and lets you know about them.

Most of the same limits that GeoAssembler and GeoLinker have are still the same in Concept. This version of Concept is designed to run on a normal Wheels system with just the minimum required ram expansion. However, Concept has a buffer for the .rel file object code that is roughly about 200 bytes larger than GeoProgrammer uses. This may not sound like much, but there are times when you'll just barely exceed the buffer limit and wish there was just a touch more. The symbol limits are the same. You can have 20 characters in a symbol, but only the first 8 count.

If you're running the assembler in 40 column mode, you'll notice a lot of screen activity. This is where the symbol tables are being built. The second half of the tables appear in the foreground screen memory. This was originally done during the development of Concept for debugging reasons. The intention was to color the screen so you couldn't see this. However, it was left alone because it's handy to see how close you're getting to the symbol table limit. At the bottom of the screen, it says "Assembling". To the left of this is a dash mark. If the screen activity continues down to this bottom line and touches this dash mark, the symbol table will be full. You'll have a visible idea now of how close you are to this limit. In 80 column mode, this doesn't show up. So, from time to time, you might want to use 40 column mode for debugging purposes.

Concept+

The integrated assembler/linker package for Wheels 64 and Wheels 128 using a SuperCPU with SuperRAM.
Operates in both 40 and 80 column modes on the 128.

This section pertains only to the extra features of Concept+. Be sure to read the previous section about Concept since everything supported by Concept is also supported by Concept+.

THINGS YOU NEED TO KNOW ABOUT CONCEPT+

The .rel files that are generated by the assembler are not compatible with the linker in Concept or GeoLinker. Likewise, the .rel files generated by Concept and GeoAssembler are not compatible with the linker in Concept+. The biggest reason is due to the difference in symbol sizes. Concept allows you to use symbols and labels up to 20 characters long, but only the first 8 characters are actually used. When Concept creates a .rel file, only the first 8 characters of each symbol are stored. Concept+ stores all 20 characters.

This doesn't mean that you can't share your source code between the two systems. As long as you keep within the guidelines and limits of Concept, you can freely exchange source code between them. This means that you must make sure all your symbols are unique within the first 8 characters. Once you decide to utilize the full capability of Concept+, then you'll lose the ability to exchange source code between the two. Of course, that's the whole idea...take advantage of this new software!

You might also find yourself taking advantage of the fact that Concept+ can have more than 10 .rel files in each .mod. It can also have more local labels within each routine. The .lnk file can span multiple pages whereas Concept's .lnk file must all fit onto the first page. You can add many, many more macros. You can have so many macros, you could probably write a new programming language utilizing macros. You can have more symbols in each file. I doubt you'll get the "symbol table overflow" error with Concept+.

Concept+ also doesn't create a .dbg file, currently. GeoDebugger can't handle the 20-character symbols. Perhaps we'll get a new debugger eventually. In the meantime, you can still use GeoDebugger, but without showing the symbol names that are provided by the .dbg file.

NEW ASSEMBLER DIRECTIVES

We've got 13 new directives for use in the assembler. They are:

.long	works like .word but generates 3 bytes
.lword	works like .word but generates 4 bytes
.wordr	works like .word but with the two bytes in reverse order.
.longr	works like .long but with the three bytes in reversed order.
.lwordr	works like .lword but with the four bytes in reversed order.
.hex	generates bytes from hex data.
.pet	same as .byte except ascii characters are converted to petascii.
.scode	same as .byte except ascii characters are converted to cbm screencode.
.org	establish a new value for the program counter.
.raw	similar to .include except no assembly is done.
.6502	allow only 6502 opcodes and addressing modes.
.65c02	allow only 65C02 opcodes and addressing modes.
.65816	(the default) allow all 65816 opcodes, addressing modes, and aliases.

In the case of .lword and .lwordr, the most significant byte will always be a zero since results are always a maximum of 24-bits. Even though the fourth byte is wasted, a table containing 4-byte values is usually easier to index into than a table of 3-byte values.

With the .hex directive, you can enter hex data without using the leading "$" character and also without a comma separating each hex byte. Here's an example:

```
    .hex    760F12DE63E0    ;this is a .hex example.
```

The line must include two hex characters for each byte. If an odd number of characters is entered, the assembler will report an error message. Also, you can only use valid hex characters. You can use your choice of upper or lower case, it doesn't matter. The only other thing that can be on a .hex line is a comment beginning with a semi-colon. The .hex directive should be handy for entering large amounts of raw data.

THE .ORG DIRECTIVE

The .org directive requires you to enter an address such as this:

```
    .org    $4000
```

When the assembler encounters the .org directive, it will begin a section that will establish values for any global labels found in this section. When it encounters the next .psect, .ramsect, or .zsect directive, the current .org section will end. The .org section will load right along with your .psect code and not at the address where it is defined to be. The difference is that the global labels found within the .org section will have addresses based on the value given as the parameter to the .org directive.

Creating an .org section is most likely to be used for assembling code that you will copy to another portion of memory. In the above example, the code is intended to be copied to address $4000. If your application requires different chunks of code to appear in different areas of memory, the .org directive is handy because it allows you to assemble these sections into one file and then your program can move them to where they belong. Here's a small example of how this could work:

```
MainInitCode:
    LoadW   r0,#extraCode
    LoadW   r1,#$4000
    LoadW   r2,#(endExtra-extraCode)
    jsr     MoveData
    . . .
;Then this can call the routine...
    LoadB   topArea,#32
    LoadB   botArea,#127
    LoadW   leftArea,#64
    LoadW   rightArea,#255
    jsr     ClearArea
    . . .

extraCode:
    .org    $4000
ClearArea:
    lda     #0
    jsr     SetPattern
    MoveB   topArea,r2L
    MoveB   botArea,r2H
    MoveW   leftArea,r3
    MoveW   rightArea,r4
```

```
    jmp     Rectangle

topArea:
    .block  1
botArea
    .block  1
leftArea
    .block  2
rightArea
    .block  2

    .psect
endExtra:
```

The .org directive can come in handy for some of your projects. In the past, I've had to use some equates to accomplish the same thing. I would have written the above code as follows:

```
EC==$4000

MainInitCode:
    LoadW   r0,#extraCode
    LoadW   r1,#$4000
    LoadW   r2,#(endExtra-extraCode)
    jsr     MoveData
    . . .
;Then this can call the routine...
    LoadB   EC+topArea-ClearArea,#32
    LoadB   EC+botArea-ClearArea,#127
    LoadW   EC+leftArea-ClearArea,#64
    LoadW   EC+rightArea-ClearArea,#255
    jsr     ClearArea
    . . .

extraCode:
ClearArea:
    lda     #0
    jsr     SetPattern
    MoveB   EC+topArea-ClearArea,r2L
    MoveB   EC+botArea-ClearArea,r2H
    MoveW   EC+leftArea-ClearArea,r3
    MoveW   EC+rightArea-ClearArea,r4
    jmp     Rectangle
```

```
topArea:
    .block  1
botArea
    .block  1
leftArea
    .block  2
rightArea
    .block  2
endExtra:
```

This always worked, but it wasn't as handy. And I could easily forget to do it correctly which could introduce a bug.

Things you must remember when using .org:

* You can't have a .ramsect or .zsect area within an .org section.
* It's OK to use .header within an .org section.
* Branch instructions cannot branch outside of an .org section. Think of an .org section as a large routine that you cannot branch out of. This includes the BRL instruction. It can be used to branch to an address within the .org section, but not outside of it. You can use the JMP instruction to jump to an outside address.
* A single .org section cannot span multiple .rel files. It can span multiple assembler source files as long as each source file is .include'd within the same .rel file.
* An .org section must be visible during pass 1 and pass 2. Do not put it inside of a ".if Pass1, .endif" section.

THE .RAW DIRECTIVE

The .raw directive is similar to the .include directive except no assembly work is done. The file will be inserted as raw data. This is handy for adding data to your program. Here's an example:

```
.raw    help.txt
```

This would insert all the data from the file "help.txt" into your program at the point where the .raw directive is used.

If the file is a non-GEOS type file, every byte from the file will be inserted. If it's a GEOS type file and it's also a sequential format type, all the data except for the header block will be inserted. If the file is a VLIR type file, then the default is to insert the data from record #0. If you want a different record to be used, then simply add a comma followed by the record number to the filename:

```
.raw    LW_Roma,10
```

The above example would insert record #10 from the LW_Roma file into your program. As you can guess, this is handy for inserting a font into your code. In this case, LW_Roma point size 10 is inserted.

Now, if you only wanted the header block of a GEOS type file to be inserted, just add an "h" after the filename:

```
.raw    GEOWRITE 128,h
```

This would insert the 254 data bytes from the header block of GEOWRITE 128 into your program.

Now, maybe you don't want an entire file or VLIR record to be inserted. You can specify an optional starting byte and an ending byte using an "s" or "e" followed by a value. For instance, let's say we wanted just the icon data from GEOWRITE 128. The following will do this for us:

```
.raw    GEOWRITE 128,h,s2,e65
```

Since the first data byte begins at byte 2 in the sector, the above example will fetch bytes 4-67 from the sector containing the header block of GEOWRITE 128.

If you do not specify a starting byte, then the starting byte will default to byte 0 of the file. If you do not specify an ending byte, it will default to the last byte of the file (or

header block).

You can also add a "p" parameter. The value added to "p" is allowed to be greater than the ending byte value. After the data is inserted, this will add additional zero bytes until the value given for "p" is reached.

```
.raw    GEOWRITE 128,h,s0,e157,p253
```

The above will load the first 158 bytes from the header block of GEOWRITE 128 and will add zero bytes to fill out a total of 254 bytes of data. This particular example will load everything except for the comment field and will zero out the comment field. In this case, we use both the "e" and "p" parameters. If we had omitted the "e" parameter, the entire header block would have been loaded as-is since it consists of 254 bytes and no padding would have been needed.

Think of the above example the same way as the following:

```
.raw    GEOWRITE 128,h,s0,e157
.block  96
```

That would produce the same results.

I'm sure we will all find a use for the .raw directive once in awhile. I know I've needed it in the past simply to include font data. It's no fun coding a font using .byte statements.

NOTE: The .raw directive must be visible in both pass 1 and pass 2 of the assembler. Do not use code such as:

```
.if     Pass1
.raw    PaintImage
.endif
```

The code will not be properly inserted if you do this.

THE PROCESSOR DIRECTIVES

There are 3 directives that instruct the assembler on which processors it should allow coding for.

.65816 This is the default mode used when the assembler begins. All available opcodes, addressing modes, and opcode aliases may be used.

.6502 This allows only 6502 opcodes and addressing modes to be used. Opcode aliases are not allowed.

.65c02 This allows only 6502 plus 65C02 opcodes and addressing modes to be used. Opcode aliases are not allowed.

If you're coding for the SuperCPU, you need not worry about using the .65816 directive in your code since this is the default mode used by the assembler. However, if you are coding for non-SuperCPU use, you might want to add the .6502 directive at the start of each source code file. By doing this, the assembler will help you to catch typing mistakes should you enter an opcode or addressing mode that the 6502 type processors do not allow.

For example, you enter:

```
lda     (r0)
```

But you really intended to use:

```
lda     (r0),y
```

In the 65816 and 65c02 modes, the assembler would allow your mistake since it's a legal addressing mode. But with the .6502 directive being used, your mistake will be flagged as an error and you'll have the opportunity to correct it. This will help prevent long debugging sessions!

This could also be handy if you have just a particular section of code that will be used for a 65C02 but the majority of the code will be for the 6502. One good example would be a disk driver used with the FD-4000 which just so happens to use a 65C02 processor. The driver could contain some code that will get transferred into the drive's ram and will be run within the drive, but the rest of the driver code remains in the computer's memory. That section of code going to the drive could begin with the .65c02 directive and then at the end of that section, you could put a .6502 directive to return the assembler to the correct mode for the rest of the driver. This particular example could also benefit from the use of the .org directive in front of the code that will be sent to the drive.

NOTE: These directives must be visible in both pass 1 and pass 2 of the assembler. Do not use code such as:

```
.if     Pass1
.6502
.endif
```

You will get unpredictable results if you do this.

SEMI-COLONS

Another improvement relates to the use of a semi-colon in byte or immediate mode instructions. Look at the following two lines:

```
lda     #';'
.byte   ';'
```

In GeoAssembler and Concept, these two lines would produce an error message since the semi-colon was seen as the start of a comment. In Concept+, this works correctly. On the other hand, the following line has always produced the correct code:

```
.byte   "This contains a semi-colon;",0
```

A semi-colon appearing within a quoted string has always worked correctly.

REFERENCING 24-BIT VALUES

Since we can work with 24-bit values in some of the 65816 "long" addressing modes, it would be handy if we could make reference to the uppermost byte in any of those values. We've always had the "less-than" and "greater-than" characters to use for referencing the low-byte or high-byte of a 16-bit value. And we've also been able to use the left and right brackets for the same purpose. I tend to use the left and right brackets myself. Now, we also have a character that can be used for the upper byte, or you might refer to it as the "bank" byte. For this, we use the left single quote character. Look at the following example:

```
imageTable==$030100        ;we can assign 24-bit values to
                           ;symbols.

    .byte   [imageTable
    .byte   ]imageTable
    .byte   ‘imageTable
```

This code will produce the following three bytes:

$00,$01,$03

You can also do the following to load r0 with a 24-bit value:

```
    lda     #[imageTable
    sta     r0+0
    lda     #]imageTable
    sta     r0+1
    lda     #‘imageTable
    sta     r0+2
```

Or a couple of macros will also do this:

```
    LoadW   r0,#imageTable
    LoadB   r0+2,#‘imageTable
```

You'll also find a new macro in the included "mac816" file that does this same job, and you would use it like this:

```
    LoadL   r0,#imageTable
```

Feel free to design your own new macros now. After all, we have a 64K macro buffer to work with, instead of the 2K buffer that Concept and GeoAssembler use.

The left single-quote character is used to fetch the bank byte from a value. To produce the left single-quote on your keyboard, you would type CMDR @.

The original 65816 assembler standard calls for the pipe "|" symbol to be used for this purpose. Or for systems that can't produce the pipe symbol, then the exclamation mark "!" symbol is supposed to be used. And then there's some assemblers that use the up-arrow "^" symbol. However, all of these symbols are already used as operators in GeoAssembler and Concept, so I had to come up with a different character and the left single-quote seems the most fitting. I could have used the forward slash "\", but I'd prefer to reserve that for future use within C-like strings.

ALIASES

Now, if you plan to code for the SuperCPU and its 65c816 processor, this is where things get a little tricky. But, I've done some programming in Concept+ to make our lives easier. Here's a simple example of how coding for the SuperCPU can get messy:

Let's say you've got the processor switched into native mode and you're code will be running within any bank other than bank 0. In this example, you would like to load the accumulator with a value from address $0030 in the current bank. The following instruction would accomplish this:

```
lda     $0030
```

However, the assembler would see this as a request to load from direct page and will generate the following two bytes of code:

```
$a5     $30
```

Direct page is always in bank 0 and that's not where we want to get our data from in this example. In order to actually load from the first page of the current bank instead of direct page we need to use absolute addressing. We could use a .byte statement such as:

```
.byte   $ad,$00,$00
```

...but that wouldn't be very handy. After all, a good assembler is supposed to make our programming easier, not more difficult. Concept+ has a number of opcode "aliases" you can use to force the assembler to generate the desired code. The difference is only in the spelling of the opcode. For instance, to force absolute addressing when using "lda", we would instead use "laa". Notice the second character has changed to an "a" for absolute. Likewise, if we want to force a load from direct page we would use "lza". The "z" stands for zero page. If you want to load from an absolute address from another bank, this is known as "long absolute" and we would use "lla" for this purpose. Think of an alias as being a built-in macro because the same functionality could have been done by creating a macro, but instead it's built into the assembler.

Most of the opcodes that use the same name for zero page, absolute, and long absolute have aliases that you can use. Another handy use for an alias is when you are operating the processor in native mode with 16-bit registers. Normally, the following code will load the accumulator with one byte:

```
lda     #25
```

Previously, it was necessary to do something like the following to get the results we wanted:

```
lda     #[25
.byte   ]25
```

That works, but is not the best way to do things. Some 65816 assemblers require you to add a directive in front of an area of code to make the assembler use the 16-bit accumulator mode. One particular assembler uses the following:

```
.16bit
lda     #25
.8bit
```

With Concept+, you can force the assembler to use 16-bit mode by using the correct alias. In this case, it's "laa" again.

```
laa     #25
```

When you look at this code, you know immediately that the line containing the "laa" is using 16-bit mode. There are times when you don't have to use the alias for this purpose. This would be when the value being loaded is larger than 255. The assembler knows to generate two bytes for the parameter in this case. The following example shows this:

```
lda     #$5000
```

Of course, if you get used to using the aliases, you can do so and your code will look more consistent throughout and you'll know for sure that the assembler will generate the intended code.

In all cases but 6 of the aliases, only the middle character is changed. This makes it easy to remember what the alias is used for. Let's look at the full list.

In each case, the middle character either represents zero page, absolute addressing, or long absolute. It can also mean the difference between 8 bits and 16 bits as in the case of the immediate mode instructions.

Original opcodes	**Available aliases**
adc	azc,aac,alc

and	azd,aad,ald
asl	azl,aal
bit	bzt,bat
cmp	czp,cap,clp
cpx	czx,cax
cpy	czy,cay
dec	dzc,dac
eor	ezr,ear,elr
inc	izc,iac
lda	lza,laa,lla
ldx	lzx,lax
ldy	lzy,lay
lsr	lzr,lar
ora	oza,oaa,ola
rol	rzl,ral
ror	rzr,rar
sbc	szc,sac,slc
sta	sza,saa,sla
stx	szx,sax
sty	szy,say
stz	szz,saz

The following aliases are the six that have the last character changed instead of the middle one.

dec a	dea
inc a	ina
jmp	jml
jsr	jsl
trb	trz,tra
tsb	tsz,tsa

For the above 6 aliases, dea, ina, jml, and jsl are used since they are already in wide use in other assemblers. You would use jml for a jmp long and jsl for jsr long. In the case of trb and tsb, the middle character couldn't be changed since both the first and third characters are the same.

The more you use the aliases, the more you will appreciate them. Let's say you are operating in bank 3 and want to use absolute,x addressing to load some values from address $4000 in bank 0. Normally, you would use the following code:

```
lda     $004000,x
```

In reality, this is the same as "lda $4000,x". And that's how the assembler will treat it. The following code will be generated:

$bd,$00,$40

But this will load data from the current bank instead of bank 0 (unless your code has changed the data bank register). So, the answer is to use an alias.

```
    lla     $004000,x
```

This will generate the correct "absolute long indexed,x" addressing mode and produce the following bytes:

$bf,$00,$40,$00

Likewise, if you wanted to jsr from bank 3 to bank 0, you can't use "jsr $004000" because this would jsr to $4000 in the current bank. Instead, you use "jsl $004000" to force the assembler to use the jsr long mode. If you were going the other way, "jsr $034000" would work correctly since the assembler sees that you're using an address greater than 64K. But in this case, it's still wise to use jsl since you'll know when you look at the code that you're performing a jsr long.

Another reason to use the aliases is when you are assembling code from one source file, but using symbols that are defined in other source files. In these cases, the assembler has no clue as to the actual values of the symbols. It will be up to the linker to resolve the values, but in the meantime, the assembler needs to know exactly how many bytes to establish for each instruction, and then the linker fills in the values.

Let's say that one source code file establishes a particular symbol value, but you're referencing it from another file. One file might contain:

```
IMAGE_DATA==$203000
```

And then in another file, we use the following code:

```
    lda     IMAGE_DATA,x
```

Since IMAGE_DATA is defined in one source file and not in the other, it will be up to the linker to resolve the value where "lda" is being used. But in the meantime, the assembler has no clue that IMAGE_DATA is a 24-bit value. So, we need to use an alias here:

```
    lla     IMAGE_DATA,x
```

And the assembler will create the correct code for us and the linker will be able to fill in the correct value at link time.

Once again, aside from the six opcodes, dec, inc, jmp, jsr, trb, and tsb, all the aliases have just the middle character altered. In each case, the aliases use a z, a, or l for the middle character.

z is used for 1 byte or zero page (direct page)
a is used for 2 bytes or absolute addressing
l is used for 3 bytes or long addressing.

Some addressing modes have no alias since they don't need one. For instance, "clc" has no need for an alias. And neither does "lda ($10)" since that will always use a zero page address. If you try to use "lda ($1000)" the assembler will report an error message, as it normally would. Likewise, if you try to use "laa ($10)", the assembler will also report an error message.

Here's a table of all the available aliases.

ADC

AZC	$00	65	Direct page
AAC	#$0000	69	Immediate 16-bit
AAC	$0000	6D	Absolute
ALC	$000000	6F	Absolute long
AZC	$00,x	75	DP indexed,x
AAC	$0000,x	7D	Absolute indexed,x
ALC	$000000,x	7F	Abs long indexed,x

AND

AZD	$00	25	Direct page
AAD	#$0000	29	Immediate 16-bit
AAD	$0000	2D	Absolute
ALD	$000000	2F	Absolute long
AZD	$00,x	35	DP indexed,x
AAD	$0000,x	3D	Absolute indexed,x
ALD	$000000,x	3F	Abs long indexed,x

ASL

AZL	$00	06	Direct page
AAL	$0000	0E	Absolute
AZL	$00,x	16	DP indexed,x
AAL	$0000,x	1E	Absolute indexed,x

BIT

BZT	$00	24	Direct page
BAT	$0000	2C	Absolute
BZT	$00,x	34	DP indexed,x
BAT	$0000,x	3C	Absolute indexed,x
BAT	#$0000	89	Immediate 16-bit

CMP

CZP	$00	C5	Direct page
CAP	#$0000	C9	Immediate 16-bit
CAP	$0000	CD	Absolute
CLP	$000000	CF	Absolute long
CZP	$00,x	D5	DP indexed,x
CAP	$0000,x	DD	Absolute indexed,x
CLP	$000000,x	DF	Abs long indexed,x

CPX

CAX	#$0000	E0	Immediate 16-bit
CZX	$00	E4	Direct page
CAX	$0000	EC	Absolute

CPY

CAY	#$0000	C0	Immediate 16-bit
CZY	$00	C4	Direct page
CAY	$0000	CC	Absolute

DEC

DEA		3A	Accumulator
DZC	$00	C6	Direct page
DAC	$0000	CE	Absolute
DZC	$00,x	D6	DP indexed,x
DAC	$0000,x	DE	Absolute indexed,x

EOR

EZR	$00	45	Direct page
EAR	#$0000	49	Immediate 16-bit
EAR	$0000	4D	Absolute
ELR	$000000	4F	Absolute long
EZR	$00,x	55	DP indexed,x
EAR	$0000,x	5D	Absolute indexed,x
ELR	$000000,x	5F	Abs long indexed,x

INC

INA		1A	Accumulator
IZC	$00	E6	Direct page
IAC	$0000	EE	Absolute
IZC	$00,x	F6	DP indexed,x
IAC	$0000,x	FE	Absolute indexed,x

JMP

JML	$000000	5C	Absolute long

JSR

JSL	$000000	22	Absolute long

LDA

LZA	$00	A5	Direct page
LAA	#$0000	A9	Immediate 16-bit
LAA	$0000	AD	Absolute
LLA	$000000	AF	Absolute long
LZA	$00,x	B5	DP indexed,x
LAA	$0000,x	BD	Absolute indexed,x
LLA	$000000,x	BF	Abs long indexed,x

LDX

LAX	#$0000	A2	Immediate 16-bit
LZX	$00	A6	Direct page
LAX	$0000	AE	Absolute
LZX	$00,y	B6	DP indexed,y
LAX	$0000,y	BE	Absolute indexed,y

LDY

LAY	#$0000	A0	Immediate 16-bit
LZY	$00	A4	Direct page
LAY	$0000	AC	Absolute
LZY	$00,x	B4	DP indexed,x
LAY	$0000,x	BC	Absolute indexed,x

LSR

LZR	$00	46	Direct page
LAR	$0000	4E	Absolute
LZR	$00,x	56	DP indexed,x
LAR	$0000,x	5E	Absolute indexed,x

ORA

OZA	$00	05	Direct page
OAA	#$0000	09	Immediate 16-bit
OAA	$0000	0D	Absolute
OLA	$000000	0F	Absolute long
OZA	$00,x	15	DP indexed,x
OAA	$0000,x	1D	Absolute indexed,x
OLA	$000000,x	1F	Abs long indexed,x

ROL

RZL	$00	26	Direct page
RAL	$0000	2E	Absolute
RZL	$00,x	36	DP indexed,x
RAL	$0000,x	3E	Absolute indexed,x

ROR

RZR	$00	66	Direct page
RAR	$0000	6E	Absolute
RZR	$00,x	76	DP indexed,x
RAR	$0000,x	7E	Absolute indexed,x

SBC

SZC	$00	E5	Direct page
SAC	#$0000	E9	Immediate 16-bit
SAC	$0000	ED	Absolute
SLC	$000000	EF	Absolute long
SZC	$00,x	F5	DP indexed,x
SAC	$0000,x	FD	Absolute indexed,x
SLC	$000000,x	FF	Abs long indexed,x

STA

SZA	$00	85	Direct page
SAA	$0000	8D	Absolute
SLA	$000000	8F	Absolute long
SZA	$00,x	95	DP indexed,x
SAA	$0000,x	9D	Absolute indexed,x
SLA	$000000,x	9F	Abs long indexed,x

STX

SZX	$00	86	Direct page
SAX	$0000	8E	Absolute

STY

SZY	$00	84	Direct page
SAY	$0000	8C	Absolute

STZ

SZZ	$00	64	Direct page
SZZ	$00,x	74	DP indexed,x
SAZ	$0000	9C	Absolute
SAZ	$0000,x	9C	Absolute indexed,x

TRB

TRZ	$00	14	Direct page
TRA	$0000	1C	Absolute

TSB

TSZ	$00	04	Direct page
TSA	$0000	0C	Absolute

NOTES ON MVN and MVP

The memory move instructions are confusing until you've used them a few times. What's really odd is that the code we write isn't the same as what the assembler generates. But this is the way the 65816 designers wanted the assemblers to do it and this is the way Concept+ does it too.

For instance, the following code will move 16K of data bytes from $C000 in bank 0 to $8000 in bank 3.

```
   rep     #%00110000      ;16 bit a,x,y
   lax     #$c000          ;address of source.
   lay     #$8000          ;address of destination.
   laa     #$4000-1        ;total bytes minus 1.
   phb                     ;save data bank register.
   mvn     0,3             ;source is 0, dest is 3.
   plb                     ;restore data bank register.
   sep     #%00110000      ;back to 8 bit registers.
```

(the phb and plb are needed because the processor will leave the data bank register to be whatever the destination bank is)

Here's where you need to know what's going on. Let's say you put a label in front of the mvn instruction so that you can alter the destination bank.

```
 . . .
mvnSpot:
   mvn     0,3
 . . .
```

Now, you can send data to bank 4 by doing:

```
   lda     #4
   sta     mvnSpot+2
```

But wait, that's not right! But it sure looks correct. However, your application refuses to work correctly. The reason is because the assembler arranges the order of the source and destination banks OPPOSITE of what you write in your code. When you write "mvn 0,3", the assembler will generate "$54,$03,$00". This is something you need to remember if you ever use mvn or mvp and use code modifying as in the above example.

mac816 MACRO FILE

Included with this release is a macro file I put together, called "mac816". Most of the file is a variation of the geosMac file. I removed the "bra" macro since we now have a bra opcode in the 65816. Also, I added some new macros that will extend the capability of the branch instructions. How many times have you had a rather long routine and you needed to branch back from near the end to somewhere near the beginning of the routine only to get a "branch out of range" error message. In the past, you had to add an extra branch instruction in the middle of the routine to branch through. Now you can use these new macros to branch as far as you'd like. They include:

bccl	branch on carry clear long
bcsl	branch on carry set long
bvcl	branch on overflow clear long
bvsl	branch on overflow set long
beql	branch on equal long
bnel	branch not equal long
bmil	branch on minus long
bpll	branch on plus long

You would use these just like any branch instruction. You can branch to a local label within the current routine or to any global label even if the global label is in another source file. Here's a couple of examples:

```
bccl    10$
beql    SetClock
```

NEW LINKER DIRECTIVES

The linker has two new directives:

.nongeos	same as .cbm without the load address
.raw	works just like in the assembler

The .nongeos directive provides the ability to create a non-geos type file that is basically the same as using the .cbm directive, except that a load address is not inserted at the start of the file. There might be a time when you will need to create a non-geos type file without those two extra bytes at the start of it.

The .raw directive works the same way as it does in the assembler. Having this available in the linker is handy for when you want to insert raw code without having to assemble a file. I'll make use of this when I redo the installer program for The Wave. The installer contains all the files for The Wave in different VLIR records. Setting up a .lnk file to piece this all together will make it a simple task.

Here's a small example .lnk file using the .raw directive:

```
.output    Presenter       ;output filename
.header    PresntrHdr.rel  ;header .rel filename
.vlir                      ;type of structure to use
.psect     $0400           ;address of first byte

PresntrSrc.rel             ;.rel source files for resident code
PresntrInit.rel
PresntrDisp.rel
PresntrRam.rel             ;ramsect code

.mod       1
.raw                       TitleImage;screen image for startup

.mod       2
.raw                       BackImage;background screen image
```

In the above example, the files inserted into records 1 and 2 by the linker could be screen dumps that were captured by another application. Your main resident code could be used to fetch these and display them. Without the .raw directive, you would have to supply the user with 3 files to achieve the same goal. These images would then have to be loaded from separate files.

It’s OK to mix the .raw directive in amongst your other .rel files such as:

```
PresntrSrc.rel              ;.rel source files for resident code
   .raw    TitleImage       ;screen image for startup
PresntrInit.rel
PresntrDisp.rel
PresntrRam.rel              ;ramsect code
```

In this example, you could have a label at the end of the PresntrSrc source code file that indentifies the location in memory for the screen image data.

NOTE WHEN CODING:

Keep in mind that this is a 65816 assembler, but that it also works fine when coding for 6502 machines. You must remember not to use a 65816 opcode if the intended machine uses a 6502-type processor. When creating GEOS software, it's normal to use the geosMac file to support certain macros. In this file is the "bra" macro. There is also a bra opcode that works on 65C02 and 65816 processors. If you ".include" the geosMac file, the macro will be used instead of the bra opcode. A matching macro name takes precedence over an opcode. If you're coding for the 65C02 or 65816, make a copy of the geosMac file and delete the bra macro. Call the file something like "mac816" or whatever suits your taste.

Now, if you are coding for the 6502 and make a typo in your code such as the following...

```
    lda     (r0)
```

...you will introduce a bug in your program. Perhaps this was supposed to be "lda (r0),y". Concept+ will happily assemble this line of code without reporting an error because it is valid 65816 code .