Programming with Ophis

Michael Martin

Programming with Ophis
by Michael Martin

Table of Contents

Preface ..... **v**
- Why "Ophis"? v
- Getting a copy of Ophis v

1. The basics **1**
- A note on numeric notation 1
- Producing Commodore 64 programs 1
- Related commands and options 2
- Writing the actual code 3
- Assembling the code 3

2. Labels and aliases **5**
- Temporary labels 5
- Anonymous labels 5
- Aliasing 5

3. Headers, Libraries, and Macros **7**
- Header files and libraries 7
- Macros 7
  - Macro definitions 7
  - Macro invocations 8
- Example code 8

4. Character maps **9**

5. Local variables and memory segments **11**

6. Expressions **13**

7. Advanced Memory Segments **15**
- The Problem 15
- The Solution 15
- Where to go from here 16

A. Example Programs **17**
- `tutor1.oph` 17
- `tutor2.oph` 17
- `c64-1.oph` 17
- `kernal.oph` 18
- `tutor3.oph` 19
- `tutor4a.oph` 19
- `tutor4b.oph` 21
- `tutor4c.oph` 22
- `tutor5.oph` 23
- `tutor6.oph` 24
- `c64-2.oph` 26
- `tutor7.oph` 26

B. Ophis Command Reference **29**
- Command Modes 29
- Basic arguments 29
  - Numeric types 29
  - Label types 29
  - String types 30
- Compound Arguments 30
- Memory Model 30
  - Basic PC tracking 30
  - Basic Segmentation simulation 31
  - General Segmentation Simulation 31
- Macros 32
  - Defining Macros 32
  - Invoking Macros 32
  - Passing Arguments to Macros 32

Features and Restrictions of the Ophis Macro Model 32
Assembler directives.....33

Preface

The Ophis project started on a lark back in 2001. My graduate studies required me to learn Perl and Python, and I'd been playing around with Commodore 64 emulators in my spare time, so I decided to learn both languages by writing a simple cross-assembler for the 6502 chip the C-64 used in both.

The Perl version was quickly abandoned, but the Python one slowly grew in scope and power over the years, and by 2005 was a very powerful, flexible macro assembler that saw more use than I'd expect. In 2007 I finally got around to implementing the last few features I really wanted and polishing it up for general release.

Part of that process has been formatting the various little tutorials and references I'd created into a single, unified document—the one you are now reading.

Why "Ophis"?

It's actually a kind of a horrific pun. See, I was using Python at the time, and one of the things I had been hoping to do with the assembler was to produce working Apple II programs. "Ophis" is Greek for "snake", and a number of traditions also use it as the actual *name* of the serpent in the Garden of Eden. So, Pythons, snakes, and stories involving really old Apples all combined to name the assembler.

Getting a copy of Ophis

If you're reading this as part of the Ophis install, you clearly already have it. If not, as of this writing the homepage for the Ophis assembler is http://hkn.eecs.berkeley.edu/~mcmartin/ophis/. If this is out-of-date, a Web search on "Ophis 6502 assembler" (without the quotation marks) should yield its page.

Ophis is written entirely in Python and packaged using the distutils. The default installation script on Unix and Mac OS X systems should put the files where they need to go. If you are running it locally, you will need to install the `Ophis` package somewhere in your Python package path, and then put the **ophis** script somewhere in your path.

Windows users that have Python installed can use the same source distributions that the other operating systems use; **ophis.bat** will arrange the environment variables accordingly and invoke the main script.

If you are on Windows and do not have Python installed, a prepackaged system made with **py2exe** is also available. The default Windows installer will use this. In this case, all you need to do is have **ophis.exe** in your path.

Chapter 1. The basics

In this first part of the tutorial we will create a simple "Hello World" program to run on the Commodore 64. This will cover:

- How to make programs run on a Commodore 64
- Writing simple code with labels
- Numeric and string data
- Invoking the assembler

A note on numeric notation

Throughout these tutorials, I will be using a lot of both decimal and hexadecimal notation. Hex numbers will have a dollar sign in front of them. Thus, 100 = $64, and $100 = 256.

Producing Commodore 64 programs

Commodore 64 programs are stored in the `PRG` format on disk. Some emulators (such as CCS64 or VICE) can run `PRG` programs directly; others need them to be transferred to a `D64` image first.

The `PRG` format is ludicrously simple. It has two bytes of header data: This is a little-endian number indicating the starting address. The rest of the file is a single continuous chunk of data loaded into memory, starting at that address. BASIC memory starts at memory location 2048, and that's probably where we'll want to start.

Well, not quite. We want our program to be callable from BASIC, so we should have a BASIC program at the start. We guess the size of a simple one line BASIC program to be about 16 bytes. Thus, we start our program at memory location 2064 ($0810), and the BASIC program looks like this:

```
10 SYS 2064
```

We **`SAVE`** this program to a file, then study it in a debugger. It's 15 bytes long:

```
1070:0100  01 08 0C 08 0A 00 9E 20-32 30 36 34 00 00 00
```

The first two bytes are the memory location: $0801. The rest of the data breaks down as follows:

Table 1-1. BASIC program breakdown

Memory Locations	Value
$0801-$0802	2-byte pointer to the next line of BASIC code ($080C).
$0803-$0804	2-byte line number ($000A = 10).
$0805	Byte code for the **`SYS`** command.
$0806-$080A	The rest of the line, which is just the string " 2064".

Memory Locations	Value
$080B	Null byte, terminating the line.
$080C-$080D	2-byte pointer to the next line of BASIC code ($0000 = end of program).

That's 13 bytes. We started at 2049, so we need 2 more bytes of filler to make our code actually start at location 2064. These 17 bytes will give us the file format and the BASIC code we need to have our machine language program run.

These are just bytes—indistinguishable from any other sort of data. In Ophis, bytes of data are specified with the `.byte` command. We'll also have to tell Ophis what the program counter should be, so that it knows what values to assign to our labels. The `.org` (origin) command tells Ophis this. Thus, the Ophis code for our header and linking info is:

```
.byte $01, $08, $0C, $08, $0A, $00, $9E, $20
.byte $32, $30, $36, $34, $00, $00, $00, $00
.byte $00, $00
.org $0810
```

This gets the job done, but it's completely incomprehensible, and it only uses two directives—not very good for a tutorial. Here's a more complicated, but much clearer, way of saying the same thing.

```
.word $0801
.org  $0801

        .word next, 10       ; Next line and current line number
        .byte $9e," 2064",0  ; SYS 2064
next:   .word 0              ; End of program

.advance 2064
```

This code has many advantages over the first.

- It describes better what is actually happening. The `.word` directive at the beginning indicates a 16-bit value stored in the typical 65xx way (small byte first). This is followed by an `.org` statement, so we let the assembler know right away where everything is supposed to be.
- Instead of hardcoding in the value $080C, we instead use a label to identify the location it's pointing to. Ophis will compute the address of `next` and put that value in as data. We also describe the line number in decimal since BASIC line numbers generally *are* in decimal. Labels are defined by putting their name, then a colon, as seen in the definition of `next`.
- Instead of putting in the hex codes for the string part of the BASIC code, we included the string directly. Each character in the string becomes one byte.
- Instead of adding the buffer ourselves, we used `.advance`, which outputs zeros until the specified address is reached. Attempting to `.advance` backwards produces an assemble-time error.
- It has comments that explain what the data are for. The semicolon is the comment marker; everything from a semicolon to the end of the line is ignored.

Related commands and options

This code includes constants that are both in decimal and in hex. It is also possible to specify constants in octal, binary, or with an ASCII character.

- To specify decimal constants, simply write the number.
- To specify hexadecimal constants, put a $ in front.
- To specify octal constants, put a 0 (zero) in front.
- To specify binary constants, put a % in front.
- To specify ASCII constants, put an apostrophe in front.

Example: 65 = $41 = 0101 = %1000001 = 'A

There are other commands besides `.byte` and `.word` to specify data. In particular, the `.dword` command specifies four-byte values which some applications will find useful. Also, some linking formats (such as the `SID` format) have header data in big-endian (high byte first) format. The `.wordbe` and `.dwordbe` directives provide a way to specify multibyte constants in big-endian formats cleanly.

Writing the actual code

Now that we have our header information, let's actually write the "Hello world" program. It's pretty short—a simple loop that steps through a hardcoded array until it reaches a 0 or outputs 256 characters. It then returns control to BASIC with an `RTS` statement.

Each character in the array is passed as an argument to a subroutine at memory location $FFD2. This is part of the Commodore 64's BIOS software, which its development documentation calls the KERNAL. Location $FFD2 prints out the character corresponding to the character code in the accumulator.

```
        ldx #0
loop:   lda hello, x
        beq done
        jsr $ffd2
        inx
        bne loop
done:   rts

hello:  .byte "HELLO, WORLD!", 0
```

The complete, final source is available in the *tutor1.oph* file.

Assembling the code

The Ophis assembler is a collection of Python modules, controlled by a master script. On Windows, this should all have been combined into an executable file **ophis.exe**; on other platforms, the Ophis modules should be in the library and the **ophis** script should be in your path. Typing **ophis** with no arguments should give a summary of available command line options.

Table 1-2. Ophis Options

Option	Effect

Option	Effect
`-6510`	Allows the 6510 undocumented opcodes as listed in the VICE documentation.
`-65c02`	Allows opcodes and addressing modes added by the 65C02.
`-v 0`	Quiet operation. Only reports errors.
`-v 1`	Default operation. Reports files as they are loaded, and gives statistics on the final output.
`-v 2`	Verbose operation. Names each assembler pass as it runs.
`-v 3`	Debug operation: Dumps the entire IR after each pass.
`-v 4`	Full debug operation: Dumps the entire IR and symbol table after each pass.

The only options Ophis demands are an input file and an output file. Here's a sample session, assembling the tutorial file here:

```
localhost$ ophis tutor1.oph tutor1.prg -v 2
Loading tutor1.oph
Running: Macro definition pass
Running: Macro expansion pass
Running: Label initialization pass
Fixpoint failed, looping back
Running: Label initialization pass
Running: Circularity check pass
Running: Expression checking pass
Running: Easy addressing modes pass
Running: Label Update Pass
Fixpoint failed, looping back
Running: Label Update Pass
Running: Instruction Collapse Pass
Running: Mode Normalization pass
Running: Label Update Pass
Running: Assembler
Assembly complete: 45 bytes output (14 code, 29 data, 2 filler)
```

If your emulator can run `PRG` files directly, this file will now run (and print `HELLO, WORLD!`) as many times as you type **`RUN`**. Otherwise, use a `D64` management utility to put the `PRG` on a `D64`, then load and run the file off that.

Chapter 2. Labels and aliases

Labels are an important part of your code. However, since each label must normally be unique, this can lead to "namespace pollution," and you'll find yourself going through ever more contorted constructions to generate unique label names. Ophis offers two solutions to this: *anonymous labels* and *temporary labels*. This tutorial will cover both of these facilities, and also introduce the aliasing mechanism.

Temporary labels

Temporary labels are the easiest to use. If a label begins with an underscore, it will only be reachable from inside the innermost enclosing scope. Scopes begin when a `.scope` statement is encountered. This produces a new, inner scope if there is another scope in use. The `.scend` command ends the innermost currently active scope.

We can thus rewrite our header data using temporary labels, thus allowing the main program to have a label named `next` if it wants.

```
.word $0801
.org  $0801

.scope
        .word _next, 10      ; Next line and current line number
        .byte $9e," 2064",0  ; SYS 2064
_next:  .word 0              ; End of program
.scend

.advance 2064
```

Anonymous labels

Anonymous labels are a way to handle short-ranged branches without having to come up with names for the then and else branches, for brief loops, and other such purposes. To define an anonymous label, use an asterisk. To refer to an anonymous label, use a series of `+` or `-` signs. `+` refers to the next anonymous label, `++` the label after that, etc. Likewise, `-` is the most recently defined label, `--` the one before that, and so on. The main body of the Hello World program with anonymous labels would be:

```
        ldx #0
*       lda hello, x
        beq +
        jsr $ffd2
        inx
        bne -
*       rts
```

It is worth noting that anonymous labels are globally available. They are not temporary labels, and they ignore scoping restrictions.

Aliasing

Rather the reverse of anonymous labels, aliases are names given to specific memory locations. These make it easier to keep track of important constants or locations. The KERNAL routines are a good example of constants that deserve names. To assign the traditional name `chrout` to the routine at $FFD2, simply give the directive:

```
.alias chrout $ffd2
```

And change the **jsr** command to:

```
jsr chrout
```

The final version of the code is in *tutor2.oph*. It should assemble to exactly the same program as *tutor1.oph*.

Chapter 3. Headers, Libraries, and Macros

In this chapter we will split away parts of our "Hello World" program into reusable header files and libraries. We will also abstract away our string printing technique into a macro which may be invoked at will, on arbitrary strings. We will then multiply the output of our program tenfold.

Header files and libraries

The prelude to our program—the `PRG` information and the BASIC program—are going to be the same in many, many programs. Thus, we should put them into a header file to be included later. The `.include` directive will load a file and insert it as source at the designated point.

A related directive, `.require`, will include the file as long as it hasn't been included yet elsewhere. It is useful for ensuring a library is linked in.

For pre-assembled code or raw binary data, the `.incbin` directive lets you include the contents of a binary file directly in the output. This is handy for linking in pre-created graphics or sound data.

As a sample library, we will expand the definition of the `chrout` routine to include the standard names for every KERNAL routine. Our header file will then `.require` it.

We'll also add some convenience aliases for things like reverse video, color changes, and shifting between upper case/graphics and mixed case text. We'd feed those to the `chrout` routine to get their effects.

Since there have been no interesting changes to the prelude, and the KERNAL values are standard, we do not reproduce them here. (The files in question are *c64-1.oph* and *kernal.oph*.)

Macros

A macro is a way of expressing a lot of code or data with a simple shorthand. It's also usually configurable. Traditional macro systems such as C's `#define` mechanic use *textual replacement*: a macro is expanded before any evaluation or even parsing occurs.

In contrast, Ophis's macro system uses a *call by value* approach where the arguments to macros are evaluated to bytes or words before being inserted into the macro body. This produces effects much closer to those of a traditional function call. A more detailed discussion of the tradeoffs may be found in Appendix B.

Macro definitions

A macro definition is a set of statements between a `.macro` statement and a `.macend` statement. The `.macro` statement also names the macro being defined.

No global or anonymous labels may be defined inside a macro: temporary labels only persist in the macro expansion itself. (Each macro body has its own scope.)

Arguments to macros are referred to by number: the first is `_1`, the second `_2`, and so on.

Here's a macro that encapsulates the printing routine in our "Hello World" program, with an argument being the address of the string to print:

```
.macro print
        ldx #0
_loop:  lda _1, x
```

```
        beq _done
        jsr chrout
        inx
        bne _loop
_done:
.macend
```

Macro invocations

Macros may be invoked in two ways: one that looks like a directive, and one that looks like an instruction.

The most common way to invoke a macro is to backquote the name of the macro. It is also possible to use the `.invoke` command. These commands look like this:

```
`print msg
.invoke print msg
```

Arguments are passed to the macro as a comma-separated list. They must all be expressions that evaluate to byte or word values—a mechanism similar to `.alias` is used to assign their values to the `_n` names.

Example code

tutor3.oph expands our running example, including the code above and also defining a new macro `greet` that takes a string argument and prints a greeting to it. It then greets far too many targets.

Chapter 4. Character maps

Now we will close the gap between the Commodore's version of ASCII and the real one. We'll also add a time-delay routine to slow down the output. This routine isn't really of interest to us right now, so we'll add a subroutine called `delay` that executes 2,560*(accumulator) **`NOP`**s. By the time the program is finished, we'll have executed 768,000 no-ops.

There actually are better ways of getting a time-delay on the Commodore 64; we'll deal with those in Chapter 5. As a result, there isn't really a lot to discuss here. The later tutorials will be building off of *tutor4a.oph*, so you may want to get familiar with that. Note also the change to the body of the `greet` macro.

On to the topic at hand. Let's change the code to use mixed case. We defined the `upper'case` and `lower'case` aliases back in Chapter 3 as part of the standard *kernal.oph* header, so we can add this before our invocations of the `greet` macro:

```
        lda #lower'case
        jsr chrout
```

And that will put us into mixed case mode. So, now we just need to redefine the data so that it uses the mixed-case:

```
hello1:   .byte "Hello, ",0
hello2:   .byte "!", 13, 0

target1:  .byte "programmer", 0
target2:  .byte "room", 0
target3:  .byte "building", 0
target4:  .byte "neighborhood", 0
target5:  .byte "city", 0
target6:  .byte "nation", 0
target7:  .byte "world", 0
target8:  .byte "Solar System", 0
target9:  .byte "Galaxy", 0
target10: .byte "Universe", 0
```

The code that does this is in *tutor4b.oph*. If you assemble and run it, you will notice that the output is not what we want. In particular, upper and lowercase are reversed, so we have messages like `hELLO, sOLAR sYSTEM!`. For the specific case of PETSCII, we can just fix our strings, but that's less of an option if we're writing for the Apple II's character set, or targeting a game console that puts its letters in arbitrary locations. We need to remap how strings are turned into byte values. The `.charmap` and `.charmapbin` directives do what we need.

The `.charmap` directive usually takes two arguments; a byte (usually in character form) indicating the ASCII value to start remapping from, and then a string giving the new values. To do our case-swapping, we write two directives before defining any string constants:

```
.charmap 'A, "abcdefghijklmnopqrstuvwxyz"
.charmap 'a, "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Note that the `'a` constant in the second directive refers to the "a" character in the source, not in the current map.

The fixed code is in *tutor4c.oph*, and will produce the expected results when run.

An alternative is to use a `.charmapbin` directive to replace the entire character map directly. This specifies an external file, 256 bytes long, that is loaded in at that point. A binary character map for the Commodore 64 is provided with the sample programs as `petscii.map`. There are also three files, `a2normal.map`, `a2inverse.map`, and `a2blink.map` that handle the Apple II's very nonstandard character encodings.

Chapter 5. Local variables and memory segments

As mentioned in Chapter 4, there are better ways to handle waiting than just executing vast numbers of NOPs. The Commodore 64 KERNAL library includes a `rdtim` routine that returns the uptime of the machine, in 60ths of a second, as a 24-bit integer. The Commodore 64 programmer's guide available online actually has a bug in it, reversing the significance of the A and Y registers. The accumulator holds the *least* significant byte, not the most.

Here's a first shot at a better delay routine:

```
.scope
        ; data used by the delay routine
        _tmp:    .byte 0
        _target: .byte 0

delay:  sta _tmp        ; save argument (rdtim destroys it)
        jsr rdtim
        clc
        adc _tmp        ; add current time to get target
        sta _target
*       jsr rdtim
        cmp _target
        bmi -           ; Buzz until target reached
        rts
.scend
```

This works, but it eats up two bytes of file space that don't really need to be specified. Also, it's modifying data inside a program text area, which isn't good if you're assembling to a ROM chip. (Since the Commodore 64 stores its programs in RAM, it's not an issue for us here.) A slightly better solution is to use `.alias` to assign the names to chunks of RAM somewhere. There's a 4K chunk of RAM from $C000 through $CFFF between the BASIC ROM and the I/O ROM that should serve our purposes nicely. We can replace the definitions of `_tmp` and `_target` with:

```
        ; data used by the delay routine
        .alias _tmp    $C000
        .alias _target $C001
```

This works better, but now we've just added a major bookkeeping burden upon ourselves—we must ensure that no routines step on each other. What we'd really like are two separate program counters—one for the program text, and one for our variable space.

Ophis lets us do this with the `.text` and `.data` commands. The `.text` command switches to the program-text counter, and the `.data` command switches to the variable-data counter. When Ophis first starts assembling a file, it starts in `.text` mode.

To reserve space for a variable, use the .space command. This takes the form:

```
.space varname size
```

which assigns the name `varname` to the current program counter, then advances the program counter by the amount specified in `size`. Nothing is output to the final binary as a result of the `.space` command.

You may not put in any commands that produce output into a `.data` segment. Generally, all you will be using are `.org` and `.space` commands. Ophis will not complain if you use `.space` inside a `.text` segment, but this is nearly always wrong.

The final version of `delay` looks like this:

```
; DELAY routine.  Takes values from the Accumulator and pauses
; for that many jiffies (1/60th of a second).
```

```
.scope
.data
.space _tmp 1
.space _target 1

.text

delay:  sta _tmp        ; save argument (rdtim destroys it)
        jsr rdtim
        clc
        adc _tmp        ; add current time to get target
        sta _target
*       jsr rdtim
        cmp _target
        bmi -               ; Buzz until target reached
        rts
.scend
```

We're not quite done yet, however, because we have to tell the data segment where to begin. (If we don't, it starts at 0, which is usually wrong.) We add a very brief data segment to the top of our code:

```
.data
.org $C000
.text
```

This will run. However, we also ought to make sure that we aren't overstepping any boundaries. Our program text shouldn't run into the BASIC chip at $A000, and our data shouldn't run into the I/O region at $D000. The `.checkpc` command lets us assert that the program counter hasn't reached a specific point yet. We put, at the end of our code:

```
.checkpc $A000
.data
.checkpc $D000
```

The final program is available as *tutor5.oph*. Note that we based this on the all-uppercase version from the last section, not any of the charmapped versions.

Chapter 6. Expressions

Ophis permits a reasonably rich set of arithmetic operations to be done at assemble time. So far, all of our arguments and values have either been constants or label names. In this chapter, we will modify the `print` macro so that it calls a subroutine to do the actual printing. This will shrink the final code size a fair bit.

Here's our printing routine. It's fairly straightforward.

```
; PRINTSTR routine.  Accumulator stores the low byte of the address,
; X register stores the high byte.  Destroys the values of $10 and
; $11.

.scope
printstr:
        sta $10
        stx $11
        ldy #$00
_lp:    lda ($10), y
        beq _done
        jsr chrout
        iny
        bne _lp
_done:  rts
.scend
```

However, now we are faced with the problem of what to do with the `print` macro. We need to take a 16-bit value and store it in two 8-bit registers. We can use the < and > operators to take the low or high byte of a word, respectively. The `print` macro becomes:

```
.macro print
        lda #<_1
        ldx #>_1
        jsr printstr
.macend
```

Also, since BASIC uses the locations $10 and $11, we should really cache them at the start of the program and restore them at the end:

```
.data
.org $C000
.space cache 2
.text

        ; Save the zero page locations that printstr uses.
        lda $10
        sta cache
        lda $11
        sta cache+1

        ; ... main program goes here ...

        ; Restore the zero page values printstr uses.
        lda cache
        sta $10
        lda cache+1
        sta $11
```

Note that we only have to name `cache` once, but can use addition to refer to any offset from it.

Ophis supports following operations, with the following precedence levels (higher entries bind more tightly):

Table 6-1. Ophis Operators

Operators	Description
`[ ]`	Parenthesized expressions
`< >`	Byte selection (low, high)
`* /`	Multiply, divide
`+ -`	Add, subtract
`| & ^`	Bitwise OR, AND, XOR

Note that brackets, not parentheses, are used to group arithmetic operations. This is because parentheses are used for the indirect addressing modes, and it makes parsing much easier.

The code for this version of the code is in *tutor6.oph*.

Chapter 7. Advanced Memory Segments

This is the last section of the Ophis tutorial. By now we've covered the basics of every command in the assembler; in this final installment we show the full capabilities of the `.text` and `.data` commands as we produce a final set of Commodore 64 header files.

The Problem

Our `print'str` routine in *tutor6.oph* accesses memory locations $10 and $11 directly. We'd prefer to have symbolic names for them. This reprises our concerns back in Chapter 5 when we concluded that we wanted two separate program counters. Now we realize that we really need three; one for the text, one for the data, and one for the zero page data. And if we're going to allow three, we really should allow any number.

The Solution

The `.data` and `.text` commands can take a label name after them—this names a new segment. We'll define a new segment called `zp` (for "zero page") and have our zero-page variables be placed there. We can't actually use the default origin of $0000 here either, though, because the Commodore 64 reserves memory locations 0 and 1 to control its memory mappers:

```
.data zp
.org $0002
```

Now, actually, the rest of the zero page is reserved too: locations $02-$7F are used by the BASIC interpreter, and locations $80-$FF are used by the KERNAL. We don't need the BASIC interpreter, though, so we can back up all of $02-$7F at the start of our program and restore it all when we're done:

```
.scope
        ; Cache BASIC's zero page at top of available RAM.
        ldx #$7E
*       lda $01, x
        sta $CF81, x
        dex
        bne -

        jsr _main

        ; Restore BASIC's zero page and return control.

        ldx #$7E
*       lda $CF81, x
        sta $01, x
        dex
        bne -
        rts

_main:
        ; _main points at the start of the real program,
        ; which is actually outside of this scope
.scend
```

The new, improved header file is *c64-2.oph*.

Our `print'str` routine is then rewritten to declare and use a zero-page variable, like so:

```
; PRINTSTR routine.  Accumulator stores the low byte of the address,
; X register stores the high byte.  Destroys the values of $10 and
; $11.

.scope
.data zp
.space _ptr 2
.text
printstr:
        sta _ptr
        stx _ptr+1
        ldy #$00
_lp:    lda (_ptr),y
        beq _done
        jsr chrout
        iny
        bne _lp
_done:  rts
.scend
```

Also, we ought to put in an extra check to make sure our zero-page allocations don't overflow, either:

```
.data zp
.checkpc $80
```

That concludes our tour. The final source file is *tutor7.oph*.

Where to go from here

This tutorial has touched on everything that the assembler can do, but it's not really well organized as a reference. Appendix B is a better place to look up matters of syntax or consult lists of available commands.

If you're looking for projects to undertake, the Commodore 64 and Atari 2600 development communities are both very strong, and the Apple II and NES development communities are still alive and well as well. There's an annual Minigame Competition that's always looking for new entries.

Appendix A. Example Programs

This Appendix collects all the programs referred to in the course of this manual.

tutor1.oph

```
.word $0801
.org  $0801

        .word next, 10          ; Next line and current line number
        .byte $9e," 2064",0     ; SYS 2064
next:   .word 0                 ; End of program

.advance 2064

        ldx #0
loop:   lda hello, x
        beq done
        jsr $ffd2
        inx
        bne loop
done:   rts

hello:  .byte "HELLO, WORLD!", 0
```

tutor2.oph

```
.word $0801
.org  $0801

.scope
        .word _next, 10         ; Next line and current line number
        .byte $9e," 2064",0     ; SYS 2064
_next:  .word 0                 ; End of program
.scend

.advance 2064

.alias chrout $ffd2

        ldx #0
*       lda hello, x
        beq +
        jsr chrout
        inx
        bne -
*       rts

hello:  .byte "HELLO, WORLD!", 0
```

c64-1.oph

```
.word $0801
.org  $0801

.scope
        .word _next, 10         ; Next line and current line number
        .byte $9e," 2064",0     ; SYS 2064
_next:  .word 0                 ; End of program
```

```
.scend

.advance 2064

.require "kernal.oph"
```

kernal.oph

```
; KERNAL routine aliases (C64)

.alias acptr  $ffa5
.alias chkin  $ffc6
.alias chkout $ffc9
.alias chrin  $ffcf
.alias chrout $ffd2
.alias ciout  $ffa8
.alias cint   $ff81
.alias clall  $ffe7
.alias close  $ffc3
.alias clrchn $ffcc
.alias getin  $ffe4
.alias iobase $fff3
.alias ioinit $ff84
.alias listen $ffb1
.alias load   $ffd5
.alias membot $ff9c
.alias memtop $ff99
.alias open   $ffc0
.alias plot   $fff0
.alias ramtas $ff87
.alias rdtim  $ffde
.alias readst $ffb7
.alias restor $ff8a
.alias save   $ffd8
.alias scnkey $ff9f
.alias screen $ffed
.alias second $ff93
.alias setlfs $ffba
.alias setmsg $ff90
.alias setnam $ffbd
.alias settim $ffdb
.alias settmo $ffa2
.alias stop   $ffe1
.alias talk   $ffb4
.alias tksa   $ff96
.alias udtim  $ffea
.alias unlsn  $ffae
.alias untlk  $ffab
.alias vector $ff8d

; Character codes for the colors.
.alias color'0 144
.alias color'1 5
.alias color'2 28
.alias color'3 159
.alias color'4 156
.alias color'5 30
.alias color'6 31
.alias color'7 158
.alias color'8 129
.alias color'9 149
.alias color'10 150
.alias color'11 151
.alias color'12 152
```

```
.alias color'13 153
.alias color'14 154
.alias color'15 155

; ...and reverse video
.alias reverse'on 18
.alias reverse'off 146

; ...and character set
.alias upper'case 142
.alias lower'case 14
```

tutor3.oph

```
.include "c64-1.oph"

.macro print
        ldx #0
_loop:  lda _1, x
        beq _done
        jsr chrout
        inx
        bne _loop
_done:
.macend

.macro greet
        'print hello1
        'print _1
        'print hello2
.macend

        lda #147
        jsr chrout
        'greet target1
        'greet target2
        'greet target3
        'greet target4
        'greet target5
        'greet target6
        'greet target7
        'greet target8
        'greet target9
        'greet target10
        rts

hello1: .byte "HELLO, ",0
hello2: .byte "!", 13, 0

target1:  .byte "PROGRAMMER", 0
target2:  .byte "ROOM", 0
target3:  .byte "BUILDING", 0
target4:  .byte "NEIGHBORHOOD", 0
target5:  .byte "CITY", 0
target6:  .byte "NATION", 0
target7:  .byte "WORLD", 0
target8:  .byte "SOLAR SYSTEM", 0
target9:  .byte "GALAXY", 0
target10: .byte "UNIVERSE", 0
```

tutor4a.oph

```
        .include "c64-1.oph"

        .macro print
                ldx #0
        _loop:  lda _1, x
                beq _done
                jsr chrout
                inx
                bne _loop
        _done:
        .macend

        .macro greet
                lda #30
                jsr delay
                'print hello1
                'print _1
                'print hello2
        .macend

                lda #147
                jsr chrout
                'greet target1
                'greet target2
                'greet target3
                'greet target4
                'greet target5
                'greet target6
                'greet target7
                'greet target8
                'greet target9
                'greet target10
                rts

        hello1: .byte "HELLO, ",0
        hello2: .byte "!", 13, 0

        target1:  .byte "PROGRAMMER", 0
        target2:  .byte "ROOM", 0
        target3:  .byte "BUILDING", 0
        target4:  .byte "NEIGHBORHOOD", 0
        target5:  .byte "CITY", 0
        target6:  .byte "NATION", 0
        target7:  .byte "WORLD", 0
        target8:  .byte "SOLAR SYSTEM", 0
        target9:  .byte "GALAXY", 0
        target10: .byte "UNIVERSE", 0

        ; DELAY routine.  Executes 2,560*(A) NOP statements.
        delay:  tax
                ldy #00
        *       nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                iny
                bne -
                dex
```

```
        bne -
        rts
```

tutor4b.oph

```
        .include "c64-1.oph"

        .macro print
                ldx #0
        _loop:  lda _1, x
                beq _done
                jsr chrout
                inx
                bne _loop
        _done:
        .macend

        .macro greet
                lda #30
                jsr delay
                'print hello1
                'print _1
                'print hello2
        .macend

                lda #147
                jsr chrout
                lda #lower'case
                jsr chrout
                'greet target1
                'greet target2
                'greet target3
                'greet target4
                'greet target5
                'greet target6
                'greet target7
                'greet target8
                'greet target9
                'greet target10
                rts

        hello1: .byte "Hello, ",0
        hello2: .byte "!", 13, 0

        target1:  .byte "programmer", 0
        target2:  .byte "room", 0
        target3:  .byte "building", 0
        target4:  .byte "neighborhood", 0
        target5:  .byte "city", 0
        target6:  .byte "nation", 0
        target7:  .byte "world", 0
        target8:  .byte "Solar System", 0
        target9:  .byte "Galaxy", 0
        target10: .byte "Universe", 0

        ; DELAY routine.  Executes 2,560*(A) NOP statements.
        delay:  tax
                ldy #00
        *       nop
                nop
                nop
                nop
                nop
                nop
```

```
        nop
        nop
        nop
        nop
        iny
        bne -
        dex
        bne -
        rts
```

tutor4c.oph

```
.include "c64-1.oph"

.macro print
        ldx #0
_loop:  lda _1, x
        beq _done
        jsr chrout
        inx
        bne _loop
_done:
.macend

.macro greet
        lda #30
        jsr delay
        'print hello1
        'print _1
        'print hello2
.macend

        lda #147
        jsr chrout
        lda #lower'case
        jsr chrout
        'greet target1
        'greet target2
        'greet target3
        'greet target4
        'greet target5
        'greet target6
        'greet target7
        'greet target8
        'greet target9
        'greet target10
        rts

.charmap 'A, "abcdefghijklmnopqrstuvwxyz"
.charmap 'a, "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

hello1: .byte "Hello, ",0
hello2: .byte "!", 13, 0

target1:  .byte "programmer", 0
target2:  .byte "room", 0
target3:  .byte "building", 0
target4:  .byte "neighborhood", 0
target5:  .byte "city", 0
target6:  .byte "nation", 0
target7:  .byte "world", 0
target8:  .byte "Solar System", 0
target9:  .byte "Galaxy", 0
target10: .byte "Universe", 0
```

```
; DELAY routine.  Executes 2,560*(A) NOP statements.
delay:  tax
        ldy #00
*       nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        iny
        bne -
        dex
        bne -
        rts
```

tutor5.oph

```
.include "c64-1.oph"

.data
.org $C000
.text

.macro print
        ldx #0
_loop:  lda _1, x
        beq _done
        jsr chrout
        inx
        bne _loop
_done:
.macend

.macro greet
        lda #30
        jsr delay
        'print hello1
        'print _1
        'print hello2
.macend

        lda #147
        jsr chrout
        'greet target1
        'greet target2
        'greet target3
        'greet target4
        'greet target5
        'greet target6
        'greet target7
        'greet target8
        'greet target9
        'greet target10
        rts

hello1: .byte "HELLO, ",0
hello2: .byte "!", 13, 0

target1:  .byte "PROGRAMMER", 0
```

```
target2:  .byte "ROOM", 0
target3:  .byte "BUILDING", 0
target4:  .byte "NEIGHBORHOOD", 0
target5:  .byte "CITY", 0
target6:  .byte "NATION", 0
target7:  .byte "WORLD", 0
target8:  .byte "SOLAR SYSTEM", 0
target9:  .byte "GALAXY", 0
target10: .byte "UNIVERSE", 0

; DELAY routine.  Takes values from the Accumulator and pauses
; for that many jiffies (1/60th of a second).
.scope
.data
.space _tmp 1
.space _target 1

.text

delay:  sta _tmp        ; save argument (rdtim destroys it)
        jsr rdtim
        clc
        adc _tmp        ; add current time to get target
        sta _target
*       jsr rdtim
        cmp _target
        bmi -           ; Buzz until target reached
        rts
.scend

.checkpc $A000
.data
.checkpc $D000
```

tutor6.oph

```
.include "c64-1.oph"

.data
.org $C000
.space cache 2
.text

.macro print
        lda #<_1
        ldx #>_1
        jsr printstr
.macend

.macro greet
        lda #30
        jsr delay
        'print hello1
        'print _1
        'print hello2
.macend

        ; Save the zero page locations that PRINTSTR uses.
        lda $10
        sta cache
        lda $11
        sta cache+1

        lda #147
```

```
        jsr chrout
        'greet target1
        'greet target2
        'greet target3
        'greet target4
        'greet target5
        'greet target6
        'greet target7
        'greet target8
        'greet target9
        'greet target10

        ; Restore the zero page values printstr uses.
        lda cache
        sta $10
        lda cache+1
        sta $11

        rts

hello1: .byte "HELLO, ",0
hello2: .byte "!", 13, 0

target1:  .byte "PROGRAMMER", 0
target2:  .byte "ROOM", 0
target3:  .byte "BUILDING", 0
target4:  .byte "NEIGHBORHOOD", 0
target5:  .byte "CITY", 0
target6:  .byte "NATION", 0
target7:  .byte "WORLD", 0
target8:  .byte "SOLAR SYSTEM", 0
target9:  .byte "GALAXY", 0
target10: .byte "UNIVERSE", 0

; DELAY routine.  Takes values from the Accumulator and pauses
; for that many jiffies (1/60th of a second).
.scope
.data
.space _tmp 1
.space _target 1

.text

delay:  sta _tmp        ; save argument (rdtim destroys it)
        jsr rdtim
        clc
        adc _tmp        ; add current time to get target
        sta _target
*       jsr rdtim
        cmp _target
        bmi -           ; Buzz until target reached
        rts
.scend

; PRINTSTR routine.  Accumulator stores the low byte of the address,
; X register stores the high byte.  Destroys the values of $10 and
; $11.

.scope
printstr:
        sta $10
        stx $11
        ldy #$00
_lp:    lda ($10),y
        beq _done
        jsr chrout
```

```
        iny
        bne _lp
_done:  rts
.scend

.checkpc $A000
.data
.checkpc $D000
```

c64-2.oph

```
.word $0801
.org  $0801

.scope
        .word _next, 10         ; Next line and current line number
        .byte $9e," 2064",0     ; SYS 2064
_next:  .word 0                 ; End of program
.scend

.advance $0810

.require "kernal.oph"

.data zp
.org $0002

.text

.scope
        ; Cache BASIC's zero page at top of available RAM.
        ldx #$7E
*       lda $01, x
        sta $CF81, x
        dex
        bne -

        jsr _main

        ; Restore BASIC's zero page and return control.

        ldx #$7E
*       lda $CF81, x
        sta $01, x
        dex
        bne -
        rts

_main:
        ; Program follows...
.scend
```

tutor7.oph

```
.include "c64-2.oph"

.data
.org $C000
.text

.macro print
```

```
        lda #<_1
        ldx #>_1
        jsr printstr
.macend

.macro greet
        lda #30
        jsr delay
        'print hello1
        'print _1
        'print hello2
.macend

        lda #147
        jsr chrout
        'greet target1
        'greet target2
        'greet target3
        'greet target4
        'greet target5
        'greet target6
        'greet target7
        'greet target8
        'greet target9
        'greet target10

        rts

hello1: .byte "HELLO, ",0
hello2: .byte "!", 13, 0

target1:  .byte "PROGRAMMER", 0
target2:  .byte "ROOM", 0
target3:  .byte "BUILDING", 0
target4:  .byte "NEIGHBORHOOD", 0
target5:  .byte "CITY", 0
target6:  .byte "NATION", 0
target7:  .byte "WORLD", 0
target8:  .byte "SOLAR SYSTEM", 0
target9:  .byte "GALAXY", 0
target10: .byte "UNIVERSE", 0

; DELAY routine.  Takes values from the Accumulator and pauses
; for that many jiffies (1/60th of a second).
.scope
.data
.space _tmp 1
.space _target 1

.text

delay:  sta _tmp        ; save argument (rdtim destroys it)
        jsr rdtim
        clc
        adc _tmp        ; add current time to get target
        sta _target
*       jsr rdtim
        cmp _target
        bmi -           ; Buzz until target reached
        rts
.scend

; PRINTSTR routine.  Accumulator stores the low byte of the address,
; X register stores the high byte.  Destroys the values of $10 and
; $11.
```

```
.scope
.data zp
.space _ptr 2
.text
printstr:
        sta _ptr
        stx _ptr+1
        ldy #$00
_lp:    lda (_ptr),y
        beq _done
        jsr chrout
        iny
        bne _lp
_done:  rts
.scend

.checkpc $A000

.data
.checkpc $D000

.data zp
.checkpc $80
```

Appendix B. Ophis Command Reference

Command Modes

These mostly follow the *MOS Technology 6500 Microprocessor Family Programming Manual*, except for the Accumulator mode. Accumulator instructions are written and interpreted identically to Implied mode instructions.

- *Implied:* `RTS`
- *Accumulator:* `LSR`
- *Immediate:* `LDA #$06`
- *Zero Page:* `LDA $7C`
- *Zero Page, X:* `LDA $7C,X`
- *Zero Page, Y:* `LDA $7C,Y`
- *Absolute:* `LDA $D020`
- *Absolute, X:* `LDA $D000,X`
- *Absolute, Y:* `LDA $D000,Y`
- *(Zero Page Indirect, X):* `LDA ($80, X)`
- *(Zero Page Indirect), Y:* `LDA ($80), Y`
- *(Absolute Indirect):* `JMP ($A000)`
- *Relative:* `BNE loop`
- *(Absolute Indirect, X):* `JMP ($A000, X)` — Only available with 65C02 extensions
- *(Zero Page Indirect):* `LDX ($80)` — Only available with 65C02 extensions

Basic arguments

Most arguments are just a number or label. The formats for these are below.

Numeric types

- *Hex:* `$41` (Prefixed with $)
- *Decimal:* `65` (No markings)
- *Octal:* `0101` (Prefixed with zero)
- *Binary:* `%01000001` (Prefixed with %)
- *Character:* `'A` (Prefixed with single quote)

Label types

Normal labels are simply referred to by name. Anonymous labels may be referenced with strings of - or + signs (the label `-` refers to the immediate previous anonymous label, `--` the one before that, etc., while `+` refers to the next anonymous label), and the special label `^` refers to the program counter at the start of the current instruction or directive.

Normal labels are *defined* by prefixing a line with the label name and then a colon (e.g., `label:`). Anonymous labels are defined by prefixing a line with an asterisk (e.g., `*`).

Temporary labels are only reachable from inside the innermost enclosing `.scope` statement. They are identical to normal labels in every way, except that they start with an underscore.

String types

Strings are enclosed in double quotation marks. Backslashed characters (including backslashes and double quotes) are treated literally, so the string `"The man said, \"The \\ character is the backslash.\""` produces the ASCII sequence for `The man said, "The \ character is the backslash."`

Strings are generally only used as arguments to assembler directives—usually for filenames (e.g., `.include`) but also for string data (in association with `.byte`).

It is legal, though unusual, to attempt to pass a string to the other data statements. This will produces a series of words/dwords where all bytes that aren't least-significant are zero. Endianness and size will match what the directive itself indicated.

Compound Arguments

Compound arguments may be built up from simple ones, using the standard +, -, *, and / operators, which carry the usual precedence. Also, the unary operators > and <, which bind more tightly than anything else, provide the high and low bytes of 16-bit values, respectively.

Use brackets [ ] instead of parentheses ( ) when grouping arithmetic operations, as the parentheses are needed for the indirect addressing modes.

Examples:

- `$D000` evaluates to $D000
- `$D000+32` evaluates to $D020
- `$D000+$20` also evaluates to $D020
- `<$D000+32` evaluates to $20
- `>$D000+32` evaluates to $F0
- `>[$D000+32]` evaluates to $D0
- `>$D000-275` evaluates to $CE

Memory Model

In order to properly compute the locations of labels and the like, Ophis must keep track of where assembled code will actually be sitting in memory, and it strives to do this in a way that is independent both of the target file and of the target machine.

Basic PC tracking

The primary technique Ophis uses is *program counter tracking*. As it assembles the code, it keeps track of a virtual program counter, and uses that to determine where the labels should go.

In the absence of an `.org` directive, it assumes a starting PC of zero. `.org` is a simple directive, setting the PC to the value that `.org` specifies. In the simplest case, one

`.org` directive appears at the beginning of the code and sets the location for the rest of the code, which is one contiguous block.

Basic Segmentation simulation

However, this isn't always practical. Often one wishes to have a region of memory reserved for data without actually mapping that memory to the file. On some systems (typically cartridge-based systems where ROM and RAM are seperate, and the target file only specifies the ROM image) this is mandatory. In order to access these variables symbolically, it's necessary to put the values into the label lookup table.

It is possible, but inconvenient, to do this with `.alias`, assigning a specific memory location to each variable. This requires careful coordination through your code, and makes creating reusable libraries all but impossible.

A better approach is to reserve a section at the beginning or end of your program, put an `.org` directive in, then use the `.space` directive to divide up the data area. This is still a bit inconvenient, though, because all variables must be assigned all at once. What we'd really like is to keep multiple PC counters, one for data and one for code.

The `.text` and `.data` directives do this. Each has its own PC that starts at zero, and you can switch between the two at any point without corrupting the other's counter. In this way each function can have a `.data` section (filled with `.space` commands) and a `.text` section (that contains the actual code). This lets our library routines be almost completely self-contained - we can have one source file that could be `.included` by multiple projects without getting in anything's way.

However, any given program may have its own ideas about where data and code go, and it's good to ensure with a `.checkpc` at the end of your code that you haven't accidentally overwritten code with data or vice versa. If your `.data` segment *did* start at zero, it's probably wise to make sure you aren't smashing the stack, too (which is sitting in the region from $0100 to $01FF).

If you write code with no segment-defining statements in it, the default segment is `text`.

The `data` segment is designed only for organizing labels. As such, errors will be flagged if you attempt to actually output information into a `data` segment.

General Segmentation Simulation

One text and data segment each is usually sufficient, but for the cases where it is not, Ophis allows for user-defined segments. Putting a label after `.text` or `.data` produces a new segment with the specified name.

Say, for example, that we have access to the RAM at the low end of the address space, but want to reserve the zero page for truly critical variables, and use the rest of RAM for everything else. Let's also assume that this is a 6510 chip, and locations $00 and $01 are reserved for the I/O port. We could start our program off with:

```
.data
.org $200
.data zp
.org $2
.text
.org $800
```

And, to be safe, we would probably want to end our code with checks to make sure we aren't overwriting anything:

```
.data
.checkpc $800
.data zp
```

```
.checkpc $100
```

Macros

Assembly language is a powerful tool—however, there are many tasks that need to be done repeatedly, and with mind-numbing minor modifications. Ophis includes a facility for *macros* to allow this. Ophis macros are very similar in form to function calls in higher level languages.

Defining Macros

Macros are defined with the `.macro` and `.macend` commands. Here's a simple one that will clear the screen on a Commodore 64:

```
.macro clr'screen
    lda #147
    jsr $FFD2
.macend
```

Invoking Macros

To invoke a macro, either use the `.invoke` command or backquote the name of the routine. The previous macro may be expanded out in either of two ways, at any point in the source:

```
.invoke clr'screen
```

or

```
`clr'screen
```

will work equally well.

Passing Arguments to Macros

Macros may take arguments. The arguments to a macro are all of the "word" type, though byte values may be passed and used as bytes as well. The first argument in an invocation is bound to the label `_1`, the second to `_2`, and so on. Here's a macro for storing a 16-bit value into a word pointer:

```
.macro store16   ; `store16 dest, src
 lda #<_2
 sta _1
 lda #>_2
 sta _1+1
.macend
```

Macro arguments behave, for the most part, as if they were defined by `.alias` commands *in the calling context*. (They differ in that they will not produce duplicate-label errors if those names already exist in the calling scope, and in that they disappear after the call is completed.)

Features and Restrictions of the Ophis Macro Model

Unlike most macro systems (which do textual replacement), Ophis macros evaluate their arguments and bind them into the symbol table as temporary labels. This produces some benefits, but it also puts some restrictions on what kinds of macros may be defined.

The primary benefit of this "expand-via-binding" discipline is that there are no surprises in the semantics. The expression `_1+1` in the macro above will always evaluate to one more than the value that was passed as the first argument, even if that first argument is some immensely complex expression that an expand-via-substitution method may accidentally mangle.

The primary disadvantage of the expand-via-binding discipline is that only fixed numbers of words and bytes may be passed. A substitution-based system could define a macro including the line `LDA _1` and accept as arguments both `$C000` (which would put the value of memory location $C000 into the accumulator) and `#$40` (which would put the immediate value $40 into the accumulator). If you *really* need this kind of behavior, a run a C preprocessor over your Ophis source, and use `#define` to your heart's content.

Assembler directives

Assembler directives are all instructions to the assembler that are not actual instructions. Ophis's set of directives follow.

- `.advance` *address*: Forces the program counter to be *address*. Unlike the `.org` directive, `.advance` outputs zeroes until the program counter reaches a specified address. Attempting to `.advance` to a point behind the current program counter is an assemble-time error.
- `.alias` *label value*: The .alias directive assigns an arbitrary value to a label. This value may be an arbitrary argument, but cannot reference any label that has not already been defined (this prevents recursive label dependencies).
- `.byte` *arg* [, *arg*, ...]: Specifies a series of arguments, which are evaluated, and strings, which are included as raw ASCII data. The final results of these arguments must be one byte in size. Seperate constants are seperated by comments.
- `.checkpc` *address*: Ensures that the program counter is less than or equal to the address specified, and emits an assemble-time error if it is not. *This produces no code in the final binary - it is there to ensure that linking a large amount of data together does not overstep memory boundaries.*
- `.data` *[label]*: Sets the segment to the segment name specified and disallows output. If no label is given, switches to the default data segment.
- `.incbin` *filename*: Inserts the contents of the file specified as binary data. Use it to include graphics information, precompiled code, or other non-assembler data.
- `.include` *filename*: Includes the entirety of the file specified at that point in the program. Use this to order your final sources.
- `.org` *address*: Sets the program counter to the address specified. *This does not emit any code in and of itself, nor does it overwrite anything that previously existed.* If you wish to jump ahead in memory, use `.advance`.
- `.require` *filename*: Includes the entirety of the file specified at that point in the program. Unlike `.include`, however, code included with `.require` will only be inserted once. The `.require` directive is useful for ensuring that certain code libraries are somewhere in the final binary. They are also very useful for guaranteeing that macro libraries are available.

- `.space` *label size*: This directive is used to organize global variables. It defines the label specified to be at the current location of the program counter, and then advances the program counter *size* steps ahead. No actual code is produced. This is equivalent to `label: .org ^+size`.
- `.text` *[label]*: Sets the segment to the segment name specified and allows output. If no label is given, switches to the default text segment.
- `.word` *arg* [, *arg*, ...]: Like `.byte`, but values are all treated as two-byte values and stored low-end first (as is the 6502's wont). Use this to create jump tables (an unadorned label will evaluate to that label's location) or otherwise store 16-bit data.
- `.dword` *arg* [, *arg*, ...]: Like `.word`, but for 32-bit values.
- `.wordbe` *arg* [, *arg*, ...]: Like `.word`, but stores the value in a big-endian format (high byte first).
- `.dwordbe` *arg* [, *arg*, ...]: Like `.dword`, but stores the value high byte first.
- `.scope`: Starts a new scope block. Labels that begin with an underscore are only reachable from within their innermost enclosing `.scope` statement.
- `.scend`: Ends a scope block. Makes the temporary labels defined since the last `.scope` statement unreachable, and permits them to be redefined in a new scope.
- `.macro` *name*: Begins a macro definition block. This is a scope block that can be inlined at arbitrary points with `.invoke`. Arguments to the macro will be bound to temporary labels with names like `_1`, `_2`, etc.
- `.macend`: Ends a macro definition block.
- `.invoke` *label* [*argument* [, *argument* ...]]: invokes (inlines) the specified macro, binding the values of the arguments to the ones the macro definition intends to read. A shorthand for `.invoke` is the name of the macro to invoke, backquoted.

The following directives are deprecated, added for compatibility with the old Perl assembler **P65**. Use the `-d` option to Ophis to enable them.

- `.ascii`: Equivalent to `.byte`, which didn't used to be able to handle strings.
- `.code`: Equivalent to `.text`.
- `.segment`: Equivalent to `.text`, from when there was no distinction between `.text` and `.data` segments.
- `.address`: Equivalent to `.word`.
- `.link` *filename address*: Assembles the file specified as if it began at the address specified. This is generally for use in "top-level" files, where there is not necessarily a one-to-one correspondence between file position and memory position. This is equivalent to an `.org` directive followed by an `.include`. With the introduction of the `.org` directive this one is less useful (and in most cases, any `.org` statement you use will actually be at the top of the `.include`d file).