Programming with Ophis

Michael Martin

Programming with Ophis
by Michael Martin

Table of Contents

Preface

Ophis is an assembler for the 6502 microprocessor - the famous chip used in the vast majority of the classic 8-bit computers and consoles. Its primary design goals are code readability and output flexibility - Ophis has successfully been used to create programs for the Nintendo Entertainment System, the Atari 2600, and various 8-bit Commodore machines.

Ophis's syntax is noticably different from the formats traditionally used for these chips; it draws its syntactic inspiration primarily from the assemblers for more modern chips, where the role of tokens is determined more by what they're made of and their grammatical location on a line rather than their absolute position on a line. It also borrows the sophisticated methods of tracking the location of labels when writing relinkable code—Ophis expects that the final output it produces will have only a vague resemblance to the memory image when loaded. Most of the alternatives when Ophis was first designed would place instructions and data into a memory map and then dump that map.

That said, there remain many actively used 6502 assemblers out there. If you're already a seasoned 6502 assembly programmer, or want to get your old sources built again, Ophis is likely not for you—however, if you are writing new code, or are new to the chip while still having other experience, then Ophis is a tool built with you in mind.

History of the project

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 C64 used in both.

The Perl one—uncreatively dubbed "Perl65"—was quickly abandoned, but the Python one saw more work. When it came time to name it, 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.[1]

Ophis 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 Ophis 1.0 was formally released. However, Ophis was written for Python 2.1 and this became more and more untenable as time has gone by. As I started receiving patches for parts of Ophis, and as I used it for some projects of my own, it became clear that Ophis needed to be modernized and to become better able to interoperate with other toolchains. It was this process that led to Ophis 2.

After its release Ophis 2 was picked up by a number of developers work with actual hardware from the period, including prototype machines that never saw production. Some of their contributions have refined the code generators for version 2.1.

This is an updated edition of *Programming With Ophis*, including documentation for all new features introduced and expanding the examples to include simple demonstration programs for platforms besides the Commodore 64. It also includes updated versions of the *To HLL and Back* essays I wrote using Ophis and Perl65 as example languages.

Getting a copy of Ophis

As of this writing, the Ophis assembler is hosted at Github. The latest downloads and documentation will be available at http://github.com/michaelcmartin/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.

For Windows users, 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.

If you are working on a system with Python installed but to which you do not wish to install software, there is also a standalone pure-Python edition with an ophis.py script. This may be placed anywhere and running ophis.py will temporarily set the library path to point to your directory.

About the examples

Versions of the examples in this book are available from the Ophis site. Windows users will find them packaged with the distribution; all other users can get them as a separate download or pull them directly from github.

The code in this book is available in the `examples/` subdirectory, while extra examples will be in subdirectories of their own with brief descriptions. They are largely all simple "Hello world" applications, designed mainly to demonstrate how to package assembled binaries into forms that emulators or ROM loaders can use. They are not primarily intended as tutorials for writing for the platforms themselves.

Most examples will require use of *platform headers*—standardized header files that set useful constants for the target system and, if needed, contain small programs to allow the program to be loaded and run. These are stored in the `platform/` subdirectory.

Notes

[1] Ironically, cross-platform development for the Apple II is extremely difficult, and while Ophis has been very successfully used to develop code for the Commodore 64, Nintendo Entertainment System, and Atari 2600, it has yet to actually be deployed on any of the Apples which inspired its name.

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 $080C, we instead use a label to identify the location it's pointing to. Ophis will compute the address of `next` and put that value in as data. We also describe the line number in decimal since BASIC line numbers generally *are* in decimal. Labels are defined by putting their name, then a colon, as seen in the definition of `next`.
- Instead of putting in the hex codes for the string part of the BASIC code, we included the string directly. Each character in the string becomes one byte.
- Instead of adding the buffer ourselves, we used `.advance`, which outputs zeros until the specified address is reached. Attempting to `.advance` backwards produces an assemble-time error. (If we wanted to output something besides zeros, we could add it as a second argument: `.advance 2064,$FF`, for instance.)
- 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 *hello1.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.

Ophis takes a list of source files and produces an output file based on assembling each file you give it, in order. You can add a line to your program like this to name the output file:

```
.outfile "hello.prg"
```

Alternately, you can use the `-o` option on the command line. This will override any `.outfile` directives. If you don't specify any name, it will put the output into a file named `ophis.bin`.

If you are using Ophis as part of some larger toolchain, you can also make it run in *pipe mode*. If you give a dash `-` as an input file or as the output target, Ophis will use standard input or output for communication.

Table 1-2. Ophis Options

Option	Effect
`-o FILE`	Overrides the default filename for output.
`-l FILE`	Specifies an optional listing file that gives the emitted binary in human-readable form, with disassembly.
`-m FILE`	Specifies an optional map file that gives the in-source names for every label used in the program.
`-u`	Allows the 6510 undocumented opcodes as listed in the VICE documentation.
`-c`	Allows opcodes and addressing modes added by the 65C02.
`-4`	Allows opcodes and addressing modes added by the 4502. (Experimental.)
`-q`	Quiet operation. Only reports warnings and errors.
`-v`	Verbose operation. Reports files as they are loaded.

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 -v hello1.oph
Loading hello1.oph
Assembly complete: 45 bytes output (14 code, 29 data, 2 filler)
```

This will produce a file named `hello.prg`. 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. If you have access to a device like the 1541 Ultimate II, you can even load the file directly into the actual hardware.

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
```

It’s possible to have multiple temporary labels with the same name in different parts of the code. If you create a label map in those cases, you will have to look at the sourcefile location to distinguish them.

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 *hello2.oph*. It should assemble to exactly the same program as *hello1.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.

If you only wish to include part of a binary file, .incbin takes up to two optional arguments, naming the file offset at which to start reading and the number of characters to read.

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 *c64kernal.oph*.) The c64kernal.oph header is likely to be useful in your own projects, and it is available in the platform/ directory for easy inclusion.

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. A label map will trace back through macro expansions to describe were a label inside a macro body came from.)

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

hello3.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 *hello4a.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 *c64kernal.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 *hello4b.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 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 *hello4c.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`.

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 60^{th}s 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 *hello5.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.[1]

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 *hello6.oph*.

Notes

1. We could spare ourselves some trouble here and use $fb instead of $10, which BASIC does *not* use, but the example is more thorough this way.

Chapter 7. Advanced Memory Segments

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 more sophisticated set of Commodore 64 header files.

The Problem

Our `print'str` routine in *hello6.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-$8F are used by the BASIC interpreter, and locations $90-$FF are used by the KERNAL. We don't need the BASIC interpreter, though, so we can back up all of $02-$8F at the start of our program and restore it all when we're done.

In fact, since we're disablng BASIC, we can actually also swap out its ROM entirely and get a contiguous block of RAM from $0002 to $CFFF:

```
.scope
        ; Cache BASIC zero page at top of available RAM
        ldx     #$8e
*       lda     $01, x
        sta     $cf81, x
        dex
        bne     -

        ; Swap out the BASIC ROM for RAM
        lda     $01
        and     #$fe
        ora     #$06
        sta     $01

        ; Run the real program
        jsr     _main

        ; Restore BASIC ROM
        lda     $01
        ora     #$07
        sta     $01

        ; Restore BASIC zero page
        ldx     #$8e
*       lda     $cf81, x
        sta     $01, x
        dex
        bne     -

        ; Back to BASIC
        rts

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

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
```

The final source file is *hello7.oph*.

Chapter 8. Platform-Specific Techniques

Ophis is intended to produce cross-assembled binaries that will run in a variety of contexts. The expectation is that most users will be writing for emulated versions of hardware from when the 6502 chip was current, and producing files either for those emulators or for devices that will transfer the results to real hardware. This chapter describes the support routines and examples to make those tasks easier.

The Commodore 64 and VIC-20

In a real sense, the Commodore 64 is the "native" target platform for Ophis. It was the first platform targeted and it's the one that has received the most additional support. It's also one where the developer needs to take the most care about exactly what kind of program they are writing.

- `c64kernal.oph` actually defines no code. It merely sets up the customary names for the KERNAL jump table routines so that you may refer to routines like `chrout` and `rdtim` by name.
- `c64header.oph` is an absolutely minimal C64 header program; it contains the one-line BASIC program and nothing else. Smaller programs that do not require more than four bytes of zero page do not need to do any bankswitching or zero page caching and don't need any more than this. The aliases provided in `c64kernal.oph` may be useful, but are not included in this header.
- `c64_0.oph` is suitable for larger and more sophisticated programs. It is an enhancement of the header file developed in the previous chapter. It stores the saved zero page values in the RAM shadowed by the KERNAL ROM, and it also uses a different mechanism for returning to BASIC when done that is more robust in the face of self-modifying programs such as those produced by self-extracting compressed executables or onefiled multipart programs. It is used like the other header files—just include it at the top of your source file and use `RTS` to end your program—but programs that use this header file will have all of the zero page from $02-$8F and a contiguous chunk of program RAM from $0800-$CFFF.
- `libbasic64.oph` is an experimental set of macros and routines to permit the assembly programmer to make use of the software floating point routines provided by BASIC. It is, for obvious reasons, not compatible with `c64_0.oph`, because it needs to make use of BASIC's workspace and the ROM itself. If you wish to use this file you should include it near the end of your program.
- `vic20.oph` is a header that will work for the *unexpanded* VIC-20. Memory expansion slots change where BASIC programs load, and since these headers load in the machine language program in as the suffix to a BASIC program, that also changes where they are themselves loaded. There is no trickery with bankswitching ROMs in and out—the VIC-20 does not have enough RAM to gain anything from these techniques.
- `vic20x.oph` does the same, but for a VIC-20 with one or more memory expansions.

Using LIBBASIC64

The 6502's arithmetic capabilities are rather limited. To counteract this, BASICs of the era did floating point in software and gave BASIC programmers the full suite of arithmetic operations. These operations are largely unavailable to machine language programmers.

The `libbasic64.oph` library is an attempt to address this. It is currently considered highly experimental, but initial results are very promising.

BASIC stores floating point numbers in a five-byte format, but translates them into a seven-byte format to do actual work in two Floating Point Accumulators (FAC1 and FAC2). Ophis will let you specify 5-byte constants with the `.cbmfloat` directive, which takes a string and produces the requisite five-byte value.

The floating point functions in BASIC all operate on FAC1 and are relatively reliable. The functions `abs_fac1`, `atn_fac1`, `cos_fac1`, `exp_fac1`, `int_fac1`, `log_fac1`, `rnd_fac1`, `sgn_fac1`, `sin_fac1`, and `tan_fac1` are all provided. Routines that touch the FACs tend to be extremely finicky. This system defines a set of macros and routines to manage that for you:

- `` `f_move`` *dest, source*: Copy a five-byte floating point value from *source* to *dest*.
- `` `fp_load`` *src*: Loads FAC1 with the floating point constant specified by *src*.
- `` `fp_store`` *dest*: Saves the value of FAC1 to the named memory location.
- `` `fp_print`` *src*: Prints out the value of FAC1 to the screen. You may want to call `int_fac1` first to round it. Unlike BASIC's `PRINT` statement, this routine will not bracket the number with blanks.
- `` `fp_read`` *ptr*: Attempts to convert a string to a floating point value in FAC1, in a manner similar to BASIC's `VAL` function.
- `` `fp_add`` *operand*: Adds the operand to FAC1.
- `` `fp_subtract`` *operand*: Subtracts the operand from FAC1.
- `` `fp_multiply`` *operand*: Multiplies the operand by FAC1.
- `` `fp_divide`` *operand*: Divides FAC1 by the operand.
- `` `fp_pow`` *operand*: Raises FAC1 to the operand's power.
- `` `fp_and`` *operand*: Juggles floating point-to-integer conversions to do a bitwise AND.
- `` `fp_or`` *operand*: Likewise, but for OR.
- `jsr randomize`: Calls RND(-TI) and leaves the (useless) result in FAC1. This seeds BASIC's random number generator with the number of clock ticks since poweron.
- `jsr rnd`: Calls RND(1) and leaves the result in FAC1, providing a random number between 0 and 1.
- `jsr fac1_sign`: Loads the SGN(FAC1) into the accumulator. This will be $01 if the accumulator is positive, $00 if it is zero, and $FF if it is negative. This routine is useful for branching based on the result of a floating point computation.

Other functions are available, but their preconditions are hazier. The source file is commented with the current state of knowledge.

To see some of these functions in action, the `examples` directory includes a program `kinematics.oph`, which reads numbers in from input and computes trajectories based on them.

The Nintendo Entertainment System

The NES development community is somewhat more fragmented than the others. A skeletal `nes.oph` file is provided, but memory locations are not as consistently named. Much sample code doesn't provide aliases for control registers at all.

Conveniently creating runnable NES programs is somewhat involved. Any given product was generally burned onto several chips that were affixed to one of a large number of circuit boards. These are often referred to as "mappers" by developers because their effect is to implement various bankswitching schemes. The result is a program built out of parts, each with its own origin. A "Hello World" sample program ships with Ophis. It does not use a bankswitcher, but it does split its contents into a program chip and a graphics chip, with one of two wrapper files to knit them together into a file that other software will recognize. Samples are given for the common iNES format and the defunct UNIF format.

The Atari 2600 VCS

Of all the 8-bit development communities, the Atari developers seem to be the most cohesive. The development documents available are universal, and analysts and developers alike all use the register names in the *Stella Developer's Guide*. Ophis follows their lead, providing these names in the header `stella.oph`.

The `stella.oph` header also replicates two macros that appear in the header files distributed to budding VCS developers. They are documented in the file.

Chapter 9. The Second Step

This essay discusses how to do 16-or-more bit addition and subtraction on the 6502, and how to do unsigned comparisons properly, thus making 16-bit arithmetic less necessary.

The problem

The `ADC`, `SBC`, `INX`, and `INY` instructions are the only real arithmetic instructions the 6502 chip has. In and of themselves, they aren't too useful for general applications: the accumulator can only hold 8 bits, and thus can't store any value over 255. Matters get even worse when we're branching based on values; `BMI` and `BPL` hinge on the seventh (sign) bit of the result, so we can't represent any value above 127.

The solution

We have two solutions available to us. First, we can use the "unsigned" discipline, which involves checking different flags, but lets us deal with values between 0 and 255 instead of -128 to 127. Second, we can trade speed and register persistence for multiple precision arithmetic, using 16-bit integers (-32768 to 32767, or 0-65535), 24-bit, or more.

Multiplication, division, and floating point arithmetic are beyond the scope of this essay. The best way to deal with those is to find a math library on the web (I recommend http://www.6502.org/) and use the routines there.

Unsigned arithmetic

When writing control code that hinges on numbers, we should always strive to have our comparison be with zero; that way, no explicit compare is necessary, and we can branch simply with `BEQ/BNE`, which test the zero flag. Otherwise, we use `CMP`. The `CMP` command subtracts its argument from the accumulator (without borrow), updates the flags, but throws away the result. If the value is equal, the result is zero. (`CMP` followed by `BEQ` branches if the argument is equal to the accumulator; this is probably why it's called `BEQ` and not something like `BZS`.)

Intuitively, then, to check if the accumulator is *less than* some value, we `CMP` against that value and `BMI`. The `BMI` command branches based on the Negative Flag, which is equal to the seventh bit of `CMP`'s subtract. That's exactly what we need, for signed arithmetic. However, this produces problems if you're writing a boundary detector on your screen or something and find that 192 < 4. 192 is outside of a signed byte's range, and is interpreted as if it were -64. This will not do for most graphics applications, where your values will be ranging from 0-319 or 0-199 or 0-255.

Instead, we take advantage of the implied subtraction that `CMP` does. When subtracting, the result's carry bit starts at 1, and gets borrowed from if necessary. Let us consider some four-bit subtractions.

```
C|3210       C|3210
------       ------
1|1001    9  1|1001    9
 |0100  - 4   |1100  -12
------  ---  ------  ---
1|0101    5  0|1101   -3
```

The `CMP` command properly modifies the carry bit to reflect this. When computing A-B, the carry bit is set if A >= B, and it's clear if A < B. Consider the following two code sequences.

```
  (1)                 (2)
CMP #$C0            CMP #$C0
BMI label           BCC label
```

The code in the first column treats the value in the accumulator as a signed value, and branches if the value is less than -64. (Because of overflow issues, it will actually branch for accumulator values between $40 and $BF, even though it *should* only be doing it for values between $80 and $BF. To see why, compare $40 to $C0 and look at the result.) The second column code treats the accumulator as holding an unsigned value, and branches if the value is less than 192. It will branch for accumulator values $00-$BF.

16-bit addition and subtraction

Time to use the carry bit for what it was meant to do. Adding two 8 bit numbers can produce a 9-bit result. That 9th bit is stored in the carry flag. The `ADC` command adds the carry value to its result, as well. Thus, carries work just as we'd expect them to. Suppose we're storing two 16-bit values, low byte first, in $C100-1 and $C102-3. To add them together and store them in $C104-5, this is very easy:

```
CLC
LDA $C100
ADC $C102
STA $C104
LDA $C101
ADC $C103
STA $C105
```

Subtraction is identical, but you set the carry bit first with `SEC` (because borrow is the complement of carry—think about how the unsigned compare works if this puzzles you) and, of course, using the `SBC` instruction instead of `ADC`.

The carry/borrow bit is set appropriately to let you continue, too. As long as you just keep working your way up to bytes of ever-higher significance, this generalizes to 24 (do it three times instead of two) or 32 (four, etc.) bit integers.

16-bit comparisons

Doing comparisons on extended precision values is about the same as doing them on 8-bit values, but you have to have the value you test in memory, since it won't fit in the accumulator all at once. You don't have to store the values back anywhere, either, since all you care about is the final state of the flags. For example, here's a signed comparison, branching to `label` if the value in $C100-1 is less than 1000 ($03E8):

```
SEC
LDA $C100
SBC #$E8
LDA $C101    ; We only need the carry bit from that subtract
SBC #$03
BMI label
```

All the commentary on signed and unsigned compares holds for 16-bit (or higher) integers just as it does for the 8-bit ones.

Chapter 10. Structured Programming

This essay discusses the machine language equivalents of the basic “structured programming” concepts that are part of the “imperative” family of programming languages: if/then/else, for/next, while loops, and procedures. It also discusses basic use of variables, as well as arrays, multi-byte data types (records), and sub-byte data types (bitfields). It closes by hand-compiling pseudo-code for an insertion sort on linked lists into assembler. A complete Commodore 64 application is included as a sample with this essay.

Control constructs

Branches: `if x then y else z`

This is almost the most basic control construct. The *most* basic is `if x then y`, which is a simple branch instruction (bcc/bcs/beq/bmi/bne/bpl/bvc/bvs) past the “then” clause if the conditional is false:

```
   iny
   bne no’overflow
   inx
no’overflow:
   ;; rest of code
```

This increments the value of the y register, and if it just wrapped back around to zero, it increments the x register too. It is basically equivalent to the C statement `if ((++y)==0) ++x;`. We need a few more labels to handle else clauses as well.

```
   ;; Computation of the conditional expression.
   ;; We assume for the sake of the example that
   ;; we want to execute the THEN clause if the
   ;; zero bit is set, otherwise the ELSE
   ;; clause.  This will happen after a CMP,
   ;; which is the most common kind of ’if’
   ;; statement anyway.

   BNE else’clause

   ;; THEN clause code goes here.

   JMP end’of’if’stmt
else’clause:

   ;; ELSE clause code goes here.

end’of’if’stmt:
   ;; ... rest of code.
```

Free loops: `while x do y`

A *free loop* is one that might execute any number of times. These are basically just a combination of `if` and `goto`. For a “while x do y” loop, that executes zero or more times, you’d have code like this...

```
loop’begin:
   ;; ... computation of condition, setting zero
   ;;     bit if loop is finished...
   beq loop’done
   ;; ... loop body goes here
   jmp loop’begin
```

```
loop’done:
   ;; ... rest of program.
```

If you want to ensure that the loop body executes at least once (do y while x), just move the test to the end.

```
loop’begin:
   ;; ... loop body goes here
   ;; ... computation of condition, setting zero
   ;;     bit if loop is finished...
   bne loop’begin
   ;; ... rest of program.
```

The choice of zero bit is kind of arbitrary here. If the condition involves the carry bit, or overflow, or negative, then replace the beq with bcs/bvs/bmi appropriately.

Bounded loops: `for i = x to y do z`

A special case of loops is one where you know exactly how many times you’re going through it—this is called a *bounded* loop. Suppose you’re copying 16 bytes from $C000 to $D000. The C code for that would look something like this:

```
   int *a = 0xC000;
   int *b = 0xD000;
   int i;
   for (i = 0; i < 16; i++) { a[i] = b[i]; }
```

C doesn’t directly support bounded loops; its `for` statement is just “syntactic sugar” for a while statement. However, we can take advantage of special purpose machine instructions to get very straightforward code:

```
   ldx #$00
loop:
   lda $c000, x
   sta $d000, x
   inx
   cpx #$10
   bmi loop
```

However, remember that every arithmetic operation, including `inx` and `dex`, sets the various flags, including the Zero bit. That means that if we can make our computation *end* when the counter hits zero, we can shave off some bytes:

```
   ldx #$10
loop:
   lda #$bfff, x
   sta #$cfff, x
   dex
   bne loop
```

Notice that we had to change the addresses we’re indexing from, because x takes a slightly different range of values. The space savings is small here, and it’s become slightly more unclear. (It also hasn’t actually saved any time, because the lda and sta instructions are crossing a page boundary where they weren’t before—but if the start or end arrays began at $b020 or something this wouldn’t be an issue.) This tends to work better when the precise value of the counter isn’t used in the computation—so let us consider the NES, which uses memory location $2007 as a port to its video memory. Suppose we wish to jam 4,096 copies of the hex value $20 into the video memory. We can write this *very* cleanly, using the X and Y registers as indices in a nested loop.

```
   ldx #$10
   ldy #$00
   lda #$20
loop:
   sta $2007
   iny
   bne loop
   dex
   bne loop
```

Work through this code. Convince yourself that the `sta` is executed exactly 16*256 = 4096 times.

This is an example of a *nested* loop: a loop inside a loop. Since our internal loop didn't need the X or Y registers, we got to use both of them, which is nice, because they have special incrementing and decrementing instructions. The accumulator lacks these instructions, so it is a poor choice to use for index variables. If you have a bounded loop and don't have access to registers, use memory locations instead:

```
   lda #$10
   sta counter  ; loop 16 times
loop:
   ;; Do stuff that trashes all the registers
   dec counter
   bne loop
```

That's it! These are the basic control constructs for using inside of procedures. Before talking about how to organize procedures, I'll briefly cover the way the 6502 handles its stack, because stacks and procedures are very tightly intertwined.

The stack

The 6502 has an onboard stack in page 1. You can modify the stack pointer by storing values in X register and using `txs`; an "empty" stack is value $FF. Going into a procedure pushes the address of the next instruction onto the stack, and RTS pops that value off and jumps there. (Well, not precisely. JSR actually pushes a value that's one instruction short, and RTS loads the value, increases it by one, and THEN jumps there. But that's only an issue if you're using RTS to implement jump tables.) On an interrupt, the next instruction's address is pushed on the stack, then the process flags, and it jumps to the handler. The return from interrupt restores the flags and the PC, just as if nothing had happened.

The stack only has 256 possible entries; since addresses take two bytes to store, that means that if you call something that calls something that calls something that (etc., etc., 129 times), your computation will fail. This can happen faster if you save registers or memory values on the stack (see below).

Procedures and register saving

All programming languages are designed around the concept of procedures.[1] Procedures let you break a computation up into different parts, then use them independently. However, compilers do a lot of work for you behind the scenes to let you think this. Consider the following assembler code. How many times does the loop execute?

```
loop: ldx #$10 jsr do'stuff dex bne loop
```

The correct answer is "I don't know, but it *should* be 16." The reason we don't know is because we're assuming here that the `do'stuff` routine doesn't change the value of the X register. If it does, than all sorts of chaos could result. For major routines that aren't called often but are called in places where the register state is important, you should store the old registers on the stack with code like this:

```
do'stuff:
   pha
   txa
   pha
   tya
   pha

   ;; Rest of do'stuff goes here

   pla
   tay
   pla
   tax
   pla
   rts
```

(Remember, the last item pushed onto the stack is the first one pulled off, so you have to restore them in reverse order.) That's three more bytes on the stack, so you don't want to do this if you don't absolutely have to. If `do'stuff` actually *doesn't* touch X, there's no need to save and restore the value. This technique is called *callee-save*.

The reverse technique is called *caller-save* and pushes important registers onto the stack before the routine is called, then restores them afterwards. Each technique has its advantages and disadvantages. The best way to handle it in your own code is to mark at the top of each routine which registers need to be saved by the caller. (It's also useful to note things like how it takes arguments and how it returns values.)

Variables

Variables come in several flavors.

Global variables

Global variables are variables that can be reached from any point in the program. Since the 6502 has no memory protection, these are easy to declare. Take some random chunk of unused memory and declare it to be the global variables area. All reasonable assemblers have commands that let you give a symbolic name to a memory location—you can use this to give your globals names.

Local variables

All modern languages have some concept of "local variables", which are data values unique to that invocation of that procedure. In modern architecures, this data is stored into and read directly off of the stack. The 6502 doesn't really let you do this cleanly; I'll discuss ways of handling it in a later essay. If you're implementing a system from scratch, you can design your memory model to not require such extreme measures. There are three basic techniques.

Treat local variables like registers

This means that any memory location you use, you save on the stack and restore afterwards. This can *really* eat up stack space, and it's really slow, it's often pointless, and it has a tendency to overflow the stack. I can't recommend it. But it does let you do recursion right, if you don't need to save much memory and you aren't recursing very deep.

Procedure-based memory allocation

With this technique, you give each procedure its own little chunk of memory for use with its data. All the variables are still, technically, globals; a routine *could* interfere with another's, but the discipline of "only mess with real globals, and your own locals" is very, very easy to maintain.

This has many advantages. It's *very* fast, both to write and to run, because loading a variable is an Absolute or Zero Page instruction. Also, any procedure may call any other procedure, as long as it doesn't wind up calling itself at some point.

It has two major disadvantages. First, if many routines need a lot of space, it can consume more memory than it should. Also, this technique can require significant assembler support—you must ensure that no procedure's local variables are defined in the same place as any other procedure, and it essentially requires a full symbolic linker to do right. Ophis includes commands for *memory segmentation simulation* that automate most of this task, and make writing general libraries feasible.

Partition-based memory allocation

It's not *really* necessary that no procedure overwrite memory used by any other procedure. It's only required that procedures don't write on the memory that their *callers* use. Suppose that your program is organized into a bunch of procedures, and each fall into one of three sets:

- Procedures in set A don't call anyone.
- Procedures in set B only call procedures in set A.
- Procedures in set C only call procedures in sets A or B.

Now, each *set* can be given its own chunk of memory, and we can be absolutely sure that no procedures overwrite each other. Even if every procedure in set C uses the *same* memory location, they'll never step on each other, because there's no way to get to any other routine in set C *from* any routine in set C.

This has the same time efficiencies as procedure-based memory allocation, and, given a thoughtful design aimed at using this technique, also can use significantly less memory at run time. It's also requires much less assembler support, as addresses for variables may be assigned by hand without having to worry about those addresses already being used. However, it does impose a very tight discipline on the design of the overall system, so you'll have to do a lot more work before you start actually writing code.

Constants

Constants are "variables" that don't change. If you know that the value you're using is not going to change, you should fold it into the code, either as an Immediate operand wherever it's used, or (if it's more complicated than that) as `.byte` commands in between the procedures. This is especially important for ROM-based systems such as the NES; the NES has very little RAM available, so constants should be kept in the more plentiful ROM wherever possible.

Data structures

So far, we've been treating data as a bunch of one-byte values. There really isn't a lot you can do just with bytes. This section talks about how to deal with larger and smaller elements.

Arrays

An *array* is a bunch of data elements in a row. An array of bytes is very easy to handle with the 6502 chip, because the various indexed addressing modes handle it for you. Just load the index into the X or Y register and do an absolute indexed load. In general, these are going to be zero-indexed (that is, a 32-byte array is indexed from 0 to 31.) This code would initialize a byte array with 32 entries to 0:

```
   lda #$00
   tax
loop:
   sta array,x
   inx
   cpx #$20
   bne loop
```

(If you count down to save instructions, remember to adjust the base address so that it's still writing the same memory location.)

This approach to arrays has some limits. Primary among them is that we can't have arrays of size larger than 256; we can't fit our index into the index register. In order to address larger arrays, we need to use the indirect indexed addressing mode. We use 16-bit addition to add the offset to the base pointer, then set the Y register to 0 and then load the value with `lda (ptr),y`.

Well, actually, we can do better than that. Suppose we want to clear out 8K of ram, from $2000 to $4000. We can use the Y register to hold the low byte of our offset, and only update the high bit when necessary. That produces the following loop:

```
   lda #$00  ; Set pointer value to base ($2000)
   sta ptr
   lda #$20
   sta ptr+1
   lda #$00  ; Storing a zero
   ldx #$20  ; 8,192 ($2000) iterations: high byte
   ldy #$00  ; low byte.
loop:
   sta (ptr),y
   iny
   bne loop  ; If we haven't wrapped around, go back
   inc ptr+1 ; Otherwise update high byte
   dex       ; bump counter
   bne loop  ; and continue if we aren't done
```

This code could be optimized further; the loop prelude in particular loads a lot of redundant values that could be compressed down further:

```
   lda #$00
   tay
   ldx #$20
   sta ptr
   stx ptr+1
```

That's not directly relevant to arrays, but these sorts of things are good things to keep in mind when writing your code. Done well, they can make it much smaller and faster; done carelessly, they can force a lot of bizarre dependencies on your code and make it impossible to modify later.

Records

A *record* is a collection of values all referred to as one variable. This has no immediate representation in assembler. If you have a global variable that's two bytes and a code pointer, this is exactly equivalent to three seperate variables. You can just put one label in front of it, and refer to the first byte as `label`, the second as `label+1`, and the code pointer a `label+2`.

This really applies to all data structures that take up more than one byte. When dealing with the pointer, a 16-bit value, we refer to the low byte as `ptr` (or `label+2`, in the example above), and the high byte as `ptr+1` (or `label+3`).

Arrays of records are more interesting. There are two possibilities for these. The way most high level languages treat it is by keeping the records contiguous. If you have an array of two sixteen bit integers, then the records are stored in order, one at a time. The first is in location $1000, the next in $1004, the next in $1008, and so on. You can do this with the 6502, but you'll probably have to use the indirect indexed mode if you want to be able to iterate conveniently.

Another, more unusual, but more efficient approach is to keep each byte as a seperate array, just like in the arrays example above. To illustrate, here's a little bit of code to go through a contiguous array of 16 bit integers, adding their values to some `total` variable:

```
   ldx #$10  ; Number of elements in the array
   ldy #$00  ; Byte index from array start
loop:
   clc
   lda array, y      ; Low byte
   adc total
   sta total
   lda array+1, y    ; High byte
   adc total+1
   sta total+1
   iny               ; Jump ahead to next entry
   iny
   dex               ; Check for loop termination
   bne loop
```

And here's the same loop, keeping the high and low bytes in seperate arrays:

```
   ldx #$00
loop:
   clc
   lda lowbyte,x
   adc total
   sta total
   lda highbyte,x
   adc total+1
   sta total+1
   inx
   cpx #$10
   bne loop
```

Which approach is the right one depends on what you're doing. For large arrays, the first approach is better, as you only need to maintain one base pointer. For smaller arrays, the easier indexing makes the second approach more convenient.

Bitfields

To store values that are smaller than a byte, you can save space by putting multiple values in a byte. To extract a sub-byte value, use the bitmasking commands:

- To set bits, use the `ORA` command. `ORA #$0F` sets the lower four bits to 1 and leaves the rest unchanged.
- To clear bits, use the `AND` command. `AND #$F0` sets the lower four bits to 0 and leaves the rest unchanged.

- To reverse bits, use the `EOR` command. `EOR #$0F` reverses the lower four bits and leaves the rest unchanged.
- To test if a bit is 0, AND away everything but that bit, then see if the Zero bit was set. If the bit is in the top two bits of a memory location, you can use the BIT command instead (which stores bit 7 in the Negative bit, and bit 6 in the Overflow bit).

A modest example: Insertion sort on linked lists

To demonstrate these techniques, we will now produce code to perform insertion sort on a linked list. We'll start by defining our data structure, then defining the routines we want to write, then producing actual code for those routines. A downloadable version that will run unmodified on a Commodore 64 closes the chapter.

The data structure

We don't really want to have to deal with pointers if we can possibly avoid it, but it's hard to do a linked list without them. Instead of pointers, we will use *cursors*: small integers that represent the index into the array of values. This lets us use the many-small-byte-arrays technique for our data. Furthermore, our random data that we're sorting never has to move, so we may declare it as a constant and only bother with changing the values of `head` and the `next` arrays. The data record definition looks like this:

```
  head : byte;
  data : const int[16] = [838, 618, 205, 984, 724, 301, 249, 946,
                          925,  43, 114, 697, 985, 633, 312,  86];
  next : byte[16];
```

Exactly how this gets represented will vary from assembler to assembler. Ophis does it like this:

```
.data
.space head 1
.space next 16

.text
lb:   .byte <$838,<$618,<$205,<$984,<$724,<$301,<$249,<$946
      .byte <$925,<$043,<$114,<$697,<$985,<$633,<$312,<$086
hb:   .byte >$838,>$618,>$205,>$984,>$724,>$301,>$249,>$946
      .byte >$925,>$043,>$114,>$697,>$985,>$633,>$312,>$086
```

Doing an insertion sort

To do an insertion sort, we clear the list by setting the 'head' value to -1, and then insert each element into the list one at a time, placing each element in its proper order in the list. We can consider the lb/hb structure alone as an array of 16 integers, and just insert each one into the list one at a time.

```
procedure insertion_sort
  head := -1;
  for i := 0 to 15 do
    insert_elt i
  end
end
```

This translates pretty directly. We’ll have insert_elt take its argument in the X register, and loop with that. However, given that insert_elt is going to be a complex procedure, we’ll save the value first. The assembler code becomes:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; insertion’sort:  Sorts the list defined by head, next, hb, lb.
; Arguments:  None.
; Modifies:   All registers destroyed, head and next array sorted.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

insertion’sort:
        lda #$FF        ; Clear list by storing the terminator in ’head’
        sta head
        ldx #$0         ; Loop through the lb/hb array, adding each
insertion’sort’loop:    ; element one at a time
        txa
        pha
        jsr insert_elt
        pla
        tax
        inx
        cpx #$10
        bne insertion’sort’loop
        rts
```

Inserting an element

The pseudocode for inserting an element is a bit more complicated. If the list is empty, or the value we’re inserting goes at the front, then we have to update the value of `head`. Otherwise, we can iterate through the list until we find the element that our value fits in after (so, the first element whose successor is larger than our value). Then we update the next pointers directly and exit.

```
procedure insert_elt i
begin
   if head = -1 then begin
      head := i;
      next[i] := -1;
      return;
   end;
   val := data[i];
   if val < data[i] then begin
      next[i] := head;
      head := i;
      return;
   end;
   current := head;
   while (next[current] <> -1 and val < data[next[current]]) do
      current := next[current];
   end;
   next[i] := next[current];
   next[current] := i;
end;
```

This produces the following rather hefty chunk of code:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; insert_elt: Insert an element into the linked list.  Maintains the
;             list in sorted, ascending order.  Used by
;             insertion’sort.
; Arguments:  X register holds the index of the element to add.
; Modifies:   All registers destroyed; head and next arrays updated
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
.data
.space lbtoinsert 1
.space hbtoinsert 1
.space indextoinsert 1

.text

insert_elt:
        ldy head                        ; If the list is empty, make
        cpy #$FF                        ; head point at it, and return.
        bne insert_elt’list’not’empty
        stx head
        tya
        sta next,x
        rts
insert_elt’list’not’empty:
        lda lb,x                        ; Cache the data we’re inserting
        sta lbtoinsert
        lda hb,x
        sta hbtoinsert
        stx indextoinsert
        ldy head                        ; Compare the first value with
        sec                             ; the data.  If the data must
        lda lb,y                        ; be inserted at the front...
        sbc lbtoinsert
        lda hb,y
        sbc hbtoinsert
        bmi insert_elt’not’smallest
        tya                             ; Set its next pointer to the
        sta next,x                      ; old head, update the head
        stx head                        ; pointer, and return.
        rts
insert_elt’not’smallest:
        ldx head
insert_elt’loop:                        ; At this point, we know that
        lda next,x                      ; argument > data[X].
        tay
        cpy #$FF                        ; if next[X] = #$FF, insert arg at end.
        beq insert_elt’insert’after’current
        lda lb,y                        ; Otherwise, compare arg to
        sec                             ; data[next[X]].  If we insert
        sbc lbtoinsert                  ; before that...
        lda hb,y
        sbc hbtoinsert
        bmi insert_elt’goto’next
insert_elt’insert’after’current:        ; Fix up all the next links
        tya
        ldy indextoinsert
        sta next,y
        tya
        sta next,x
        rts                             ; and return.
insert_elt’goto’next:                   ; Otherwise, let X = next[X]
        tya                             ; and go looping again.
        tax
        jmp insert_elt’loop
```

The complete application

The full application, which deals with interfacing with CBM BASIC and handles console I/O and such, is in *structuredemo.oph*.

Notes

1. Yes, all of them. Functional languages just let you do more things with them, logic programming has implicit calls to query procedures, and object-oriented “methods” are just normal procedures that take one extra argument in secret.

Chapter 11. Pointers and Indirection

The basics of pointers versus cursors (or, at the 6502 assembler level, the indirect indexed addressing mode versus the absolute indexed ones) were covered in Chapter 10 This essay seeks to explain the uses of the indirect modes, and how to implement pointer operations with them. It does *not* seek to explain why you'd want to use pointers for something to begin with; for a tutorial on proper pointer usage, consult any decent C textbook.

The absolute basics

A pointer is a variable holding the address of a memory location. Memory locations take 16 bits to represent on the 6502: thus, we need two bytes to hold it. Any decent assembler will have ways of taking the high and low bytes of an address; use these to acquire the raw values you need. The 6502 chip does not have any simple "pure" indirect modes (except for `JMP`, which is a matter for a later essay); all are indexed, and they're indexed different ways depending on which index register you use.

The simplest example

When doing a simple, direct dereference (that is, something equivalent to the C code `c=*b;`) the code looks like this:

```
ldy #0
lda (b), y
sta c
```

Even with this simple example, there are several important things to notice.

- The variable `b` *must be on the zero page*, and furthermore, it *cannot be $FF.* All your pointer values need to be either stored on the zero page to begin with or copied there before use.
- The `y` in the `lda` statement must be y. It cannot be x (that's a different form of indirection), and it cannot be a constant. If you're doing a lot of indirection, be sure to keep your Y register free to handle the indexing on the pointers.
- The `b` variable is used alone. Statements like `lda (b+2), y` are syntactically valid and sometimes even correct: it dereferences the value next to `b` after adding y to the value therein. However, it is almost guaranteed that what you *really* wanted to do was compute `*(b+2)` (that is, take the address of b, add 2 to *that*, and dereference that value); see the next section for how to do this properly.

In nearly all cases, it is the Y-register's version (Indirect Indexed) that you want to use when you're dealing with pointers. Even though either version could be used for this example, we use the Y register to establish this habit.

Pointer arithmetic

Pointer arithmetic is an obscenely powerful and dangerous technique. However, it's the most straightforward way to deal with enormous arrays, structs, indexable stacks, and nearly everything you do in C. (C has no native array or string types primarily because it allows arbitrary pointer arithmetic, which is strong enough to handle all of those without complaint and at blazing speed. It also allows for all kinds of buffer overrun security holes, but let's face it, who's going to be cracking root on your Apple II?) There are a number of ways to implement this on the 6502. We'll deal with them in increasing order of design complexity.

The straightforward, slow way

When computing a pointer value, you simply treat the pointer as if it were a 16-bit integer. Do all the math you need, then when the time comes to dereference it, simply do a direct dereference as above. This is definitely doable, and it's not difficult. However, it is costly in both space and time.

When dealing with arbitrary indices large enough that they won't fit in the Y register, or when creating values that you don't intend to dereference (such as subtracting two pointers to find the length of a string), this is also the only truly usable technique.

The clever fast way

But wait, you say. Often when we compute a value, at least one of the operations is going to be an addition, and we're almost certain to have that value be less than 256! Surely we may save ourselves an operation by loading that value into the Y register and having the load operation itself perform the final addition!

Very good. This is the fastest technique, and sometimes it's even the most readable. These cases usually involve repeated reading of various fields from a structure or record. The base pointer always points to the base of the structure (or the top of the local variable list, or what have you) and the Y register takes values that index into that structure. This lets you keep the pointer variable in memory largely static and requires no explicit arithmetic instructions at all.

However, this technique is highly opaque and should always be well documented, indicating exactly what you think you're pointing at. Then, when you get garbage results, you can compare your comments and the resulting Y values with the actual definition of the structure to see who's screwing up.

For a case where we still need to do arithmetic, consider the classic case of needing to clear out a large chunk of memory. The following code fills the 4KB of memory between $C000 and $D000 with zeroes:

```
        lda #$C0        ; Store #$C000 in mem (low byte first)
        sta mem+1
        lda #$00
        sta mem
        ldx #$04        ; x holds number of times to execute outer loop
        tay             ; accumulator and y are both 0
loop:   sta (mem), y
        iny
        bne loop        ; Inner loop ends when y wraps around to 0
        inc mem+1       ; "Carry" from the iny to the core pointer
        dex             ; Decrement outer loop count, quit if done
        bne loop
```

Used carefully, proper use of the Y register can make your code smaller, faster, *and* more readable. Used carelessly it can make your code an unreadable, unmaintainable mess. Use it wisely, and with care, and it will be your greatest ally in writing flexible code.

What about Indexed Indirect?

This essay has concerned itself almost exclusively with the Indirect Indexed—or (Indirect), Y—mode. What about Indexed Indirect—(Indirect, X)? This is a *much* less useful mode than the Y register's version. While the Y register indirection lets you implement pointers and arrays in full generality, the X register is useful for pretty much only one application: lookup tables for single byte values.

Even coming up with a motivating example for this is difficult, but here goes. Suppose you have multiple, widely disparate sections of memory that you're watching for signals. The following routine takes a resource index in the accumulator and returns the status byte for the corresponding resource.

```
; This data is sitting on the zero page somewhere
resource_status_table: .word resource0_status, resource1_status,
                       .word resource2_status, resource3_status,
                       ; etc. etc. etc.

; This is the actual program code
.text
getstatus:
        clc   ; Multiply argument by 2 before putting it in X, so that it
        asl   ; produces a value that's properly word-indexed
        tax
        lda (resource_status_table, x)
        rts
```

Why having a routine such as this is better than just having the calling routine access resourceN_status itself as an absolute memory load is left as an exercise for the reader. That aside, this code fragment does serve as a reminder that when indexing an array of anything other than bytes, you must multiply your index by the size of the objects you want to index. C does this automatically—assembler does not. Stay sharp.

Comparison with the other indexed forms

Pointers are slow. It sounds odd saying this, when C is the fastest language around on modern machines precisely because of its powerful and extensive use of pointers. However, modern architectures are designed to be optimized for C-style code (as an example, the x86 architecture allows statements like `mov eax, [bs+bx+4*di]` as a single instruction), while the 6502 is not. An (Indirect, Y) operation can take up to 6 cycles to complete just on its own, while the preparation of that command costs additional time *and* scribbles over a bunch of registers, meaning memory operations to save the values and yet more time spent. The simple code given at the beginning of this essay—loading `*b` into the accumulator—takes 7 cycles, not counting the 6 it takes to load b with the appropriate value to begin with. If b is known to contain a specific value, we can write a single Absolute mode instruction to load its value, which takes only 4 cycles and also preserves the value in the Y register. Clearly, Absolute mode should be used whenever possible.

One might be tempted to use self-modifying code to solve this problem. This actually doesn't pay off near enough for the hassle it generates; for self-modifying code, the address must be generated, then stored in the instruction, and then the data must be loaded. Cost: 16 cycles for 2 immediate loads, 2 absolute stores, and 1 absolute load. For the straight pointer dereference, we generate the address, store it in the pointer, clear the index, then dereference that. Cost: 17 cycles for 3 immediate loads, 2 zero page stores, and 1 indexed indirect load. Furthermore, unlike in the self-modifying case, loops where simple arithmetic is being continuously performed only require repeating the final load instruction, which allows for much greater time savings over an equivalent self-modifying loop.

(This point is also completely moot for NES programmers or anyone else whose programs are sitting in ROM, because programs stored on a ROM cannot modify themselves.)

Conclusion

That's pretty much it for pointers. Though they tend to make programs hairy, and learning how to properly deal with pointers is what separates real C programmers from the novices, the basic mechanics of them are not complex. With pointers you can do efficient passing of large structures, pass-by-reference, complicated return values, and dynamic memory management—and now these wondrous toys may be added to your assembler programs, too (assuming you have that kind of space to play with).

Chapter 12. Functionals

This essay deals with indirect calls. These are the core of an enormous number of high level languages: LISP's closures, C's function pointers, C++ and Java's virtual method calls, and some implementations of the `switch` statement.

These techniques vary in complexity, and most will not be appropriate for large-scale assembler projects. Of them, however, the Data-Directed approach is the most likely to lead to organized and maintainable code.

Function Pointers

Because assembly language is totally untyped, function pointers are the same as any other sixteen-bit integer. This makes representing them really quite easy; most assemblers should permit routines to be declared simply by naming the routine as a `.word` directly.

To actually invoke these methods, copy them to some sixteen-bit location (say, `target`) and then invoking the method is a simple matter of the using an indirect jump: the `JMP (target)` instruction.

There's really only one subtlety here, and it's that the indirect jump is an indirect *jump*, not an indirect *function call*. Thus, if some function `A` makes in indirect jump to some routine, when that routine returns, it returns to whoever called `A`, not `A` itself.

There are several ways of dealing with this, but only one correct way, which is to structure your procedures so that any call to `JMP (xxxx)` occurs at the very end.

A quick digression on how subroutines work

Ordinarily, subroutines are called with `JSR` and finished with `RTS`. The `JSR` instruction takes its own address, adds 2 to it, and pushes this 16-bit value on the stack, high byte first, then low byte (so that the low byte will be popped off first).

But wait, you may object. All `JSR` instructions are three bytes long. This "return address" is in the middle of the instruction. And you would be quite right; the `RTS` instruction pops off the 16-bit address, adds one to it, and *then* sets the program counter to that value.

So it *is* possible to set up a "`JSR` indirect" kind of operation by adding two to the indirect jump's address and then pushing that value onto the stack before making the jump; however, you wouldn't want to do this. It takes six bytes and trashes your accumulator, and you can get the same functionality with half the space and with no register corruption by simply defining the indirect jump to be a one-instruction routine and `JSR`-ing to it directly. As an added bonus, that way if you have multiple indirect jumps through the same pointer, you don't need to duplicate the jump instruction.

Does this mean that abusing `JSR` and `RTS` is a dead-end, though? Not at all...

Dispatch-on-type and Data-Directed Assembler

Most of the time, you care about function pointers because you've arranged them in some kind of table. You hand it an index representing the type of your argument, or which method it is you're calling, or some other determinator, and then you index into an array of routines and execute the right one.

Writing a generic routine to do this is kind of a pain. First you have to pass a 16-bit pointer in, then you have to dereference it to figure out where your table is, then you have to do an indexed dereference on *that* to get the routine you want to run, then you need to copy it out to somewhere fixed so that you can write your jump instruction.

And making this non-generic doesn't help a whole lot, since that only saves you the first two steps, but now you have to write them out in every single indexed jump instruction. If only there were some way to easily and quickly pass in a local pointer directly...

Something, say, like the `JSR` instruction, only not for program code.

Or we could just use the `JSR` statement itself, but only call this routine at the ends of other routines, much like we were organizing for indirect jumps to begin with. This lets us set up routines that look like this:

```
jump'table'alpha:
    jsr do'jump'table
    .word alpha'0, alpha'1, alpha'2
```

Where the `alpha'x` routines are the ones to be called when the index has that value. This leaves the implementation of do'jump'table, which in this case uses the Y register to hold the index:

```
do'jump'table:
    sta _scratch
    pla
    sta _jmpptr
    pla
    sta _jmpptr+1
    tya
    asl
    tay
    iny
    lda (_jmpptr), y
    sta _target
    iny
    lda (_jmpptr), y
    sta _target+1
    lda _scratch
    jmp (_target)
```

The `TYA:ASL:TAY:INY` sequence can actually be omitted if you don't mind having your Y indices be 1, 3, 5, 7, 9, etc., instead of 0, 1, 2, 3, 4, etc. Likewise, the instructions dealing with `_scratch` can be omitted if you don't mind trashing the accumulator. Keeping the accumulator and X register pristine for the target call comes in handy, though, because it means we can pass in a pointer argument purely in registers. This will come in handy soon...

VTables and Object-Oriented Assembler

The usual technique for getting something that looks object-oriented in non-object-oriented languages is to fill a structure with function pointers, and have those functions take the structure itself as an argument. This works just fine in assembler, of course (and doesn't really require anything more than your traditional jump-indirects), but it's also possible to use a lot of the standard optimizations that languages such as C++ provide.

The most important of these is the *vtable*. Each object type has its own vtable, and it's a list of function pointers for all the methods that type provides. This is a space savings over the traditional structs-with-function-pointers approach because when you have many objects of the same class, you only have to represent the vtable once. So that all objects may be treated identically, the vtable location is traditionally fixed as being the first entry in the corresponding structure.

Virtual method invocation takes an object pointer (traditionally called `self` or `this`) and a method index and invokes the approprate method on that object. Gee, where have we seen that before?

```
sprite’vtable:
    jsr do’jump’table
    .word sprite’init, sprite’update, sprite’render
```

We mentioned before that vtables are generally the first entries in objects. We can play another nasty trick here, paying an additional byte per object to have the vtable be not merely a pointer to its vtable routine, but an actual jump instruction to it. (That is, if an object is at location X, then location X is the byte value `$4C`, representing `JMP`, location X+1 is the low byte of the vtable, and location X+2 is the high byte of the vtable.) Given that, our `invokevirtual` function becomes very simple indeed:

```
invokevirtual:
    sta this
    stx this+1
    jmp (this)
```

Which, combined with all our previous work here, takes the `this` pointer in `.AX` and a method identifier in `.Y` and invokes that method on that object. Arguments besides `this` need to be set up before the call to `invokevirtual`, probably in some global argument array somewhere as discussed back in Chapter 10.

A final reminder

We’ve been talking about all these routines as if they could be copy-pasted or hand-compiled from C++ or Java code. This isn’t really the case, primarily because “local variables” in your average assembler routines aren’t really local, so multiple calls to the same method will tend to trash the program state. And since a lot of the machinery described here shares a lot of memory (in particular, every single method invocation everywhere shares a `this`), attempting to shift over standard OO code into this format is likely to fail miserably.

You can get an awful lot of flexibility out of even just one layer of method-calls, though, given a thoughtful design. The `do’jump’table` routine, or one very like it, was extremely common in NES games in the mid-1980s and later, usually as the beginning of the frame-update loop.

If you find you really need multiple layers of method calls, though, then you really are going to need a full-on program stack, and that’s going to be several kinds of mess. That’s the topic for the final chapter.

Chapter 13. Call Stacks

All our previous work has been assuming FORTRAN-style calling conventions. In this, all procedure-local variables are actually secretly globals. This means that a function that calls itself will end up stomping on its previous values, and everything will be hideously scrambled. Various workarounds for this are covered in Chapter 10. Here, we solve the problem fully.

Recursion

A procedure in C or other similar languages declares a chunk of storage that's unique to that invocation. This chunk is just large enough to hold the return address and all the local variables, and is called the *stack frame*. Stack frames are arranged on a *call stack*; when a function is called, the stack grows with the new frame, and when that function returns, its frame is destroyed. Once the main function returns, the stack is empty.

Most modern architectures are designed to let you implement variable access like this directly, without touching the registers at all. The x86 architecture even dedicates a register to function explicitly as the *stack pointer*, and then one could read, say, the fifth 16-bit variable into the register AX with the command `MOV AX, [SP+10]`.

As we saw in Chapter 11, the 6502 isn't nearly as convenient. We'd need to keep the stack pointer somewhere on the zero page, then load the Y register with 10, then load the accumulator with an indexed-indirect call. This is verbose, keeps trashing our registers, and it's very, very slow.

So, in the spirit of programmers everywhere, we'll cheat.

Our Goals

The system we develop should have all of the following characteristics.

- It should be *intuitive to program for*. The procedure bodies should be easily readable and writable by humans, even in assembler form.
- It should be *efficient*. Variable accesses are very common, so procedures shouldn't cost much to run.
- It should allow *multiple arity* in both arguments and return values. We won't require that an unlimited amount of information be passable, but it should allow more than the three bytes the registers give us.
- It should permit *tail call elimination*, an optimization that will allow certain forms of recursion to actually not grow the stack.

Here is a system that meets all these properties.

- Reserve two bytes of the zero page for a stack pointer. At the beginning of the program, set it to the top of memory.
- Divide the remainder of Zero Page into two parts:
  - The *scratch space*, which is where arguments and return values go, and which may be scrambled by any function call, and
  - The *local area*, which all functions must restore to their initial state once finished.
- Assign to each procedure a *frame size* S, which is a maximum size on the amount of the local area the procedure can use. The procedure's variables will sit in the first S bytes of the local area.

- Upon entering the procedure, push the first S bytes of the local area onto the stack; upon exit, pop hose S bytes back on top of the local area.
- While the procedure is running, only touch the local area and the scratch space.

This meets our design criteria neatly:

- It's as intuitive as such a system will get. You have to call `init'stack` at the beginning, and you need to ensure that `save'stack` and `restore'stack` are called right. The procedure's program text can pretend that it's just referring to its own variables, just like with the old style. If a procedure doesn't call *anyone*, then it can just do all its work in the scratch space.
- It's efficient; the inside of the procedure is likely to be faster and smaller than its FORTRAN-style counterpart, because all variable references are on the Zero Page.
- Both arguments and return values can be as large as the scratch space. It's not infinite, but it's probably good enough.
- Tail call elimination is possible; just restore the stack before making the JMP to the tail call target.

The necessary support code is pretty straightforward. The stack modification routines take the size of the frame in the accumulator, and while saving the local area, it copies over the corresponding values from the scratch space. (This is because most functions will be wanting to keep their arguments around across calls.)

```
.scope
; Stack routines
.data zp
.space _sp      $02
.space _counter $01
.space fun'args $10
.space fun'vars $40

.text
init'stack:
        lda     #$00
        sta     _sp
        lda     #$A0
        sta     _sp+1
        rts

save'stack:
        sta     _counter
        sec
        lda     _sp
        sbc     _counter
        sta     _sp
        lda     _sp+1
        sbc     #$00
        sta     _sp+1
        ldy     #$00
*       lda     fun'vars, y
        sta     (_sp), y
        lda     fun'args, y
        sta     fun'vars, y
        iny
        dec     _counter
        bne -
        rts

restore'stack:
        pha
        sta     _counter
        ldy     #$00
*       lda     (_sp), y
```

```
        sta     fun’vars, y
        iny
        dec     _counter
        bne -
        pla
        clc
        adc     _sp
        sta     _sp
        lda     _sp+1
        adc     #$00
        sta     _sp+1
        rts
.scend
```

Example: Fibonnacci Numbers

About the simplest “interesting” recursive function is the Fibonacci numbers. The function fib(x) is defined as being 1 if x is 0 or 1, and being fib(x-2)+fib(x-1) otherwise.

Actually expressing it like that directly produces a very inefficient implementation, but it’s a simple demonstration of the system. Here’s code for expressing the fib function:

```
.scope
; Uint16 fib (Uint8 x): compute Xth fibonnaci number.
; fib(0) = fib(1) = 1.
; Stack usage: 3.

fib:    lda     #$03
        jsr     save’stack
        lda     fun’vars
        cmp     #$02
        bcc     _base

        dec     fun’args
        jsr     fib
        lda     fun’args
        sta     fun’vars+1
        lda     fun’args+1
        sta     fun’vars+2
        lda     fun’vars
        sec
        sbc     #$02
        sta     fun’args
        jsr     fib
        clc
        lda     fun’args
        adc     fun’vars+1
        sta     fun’args
        lda     fun’args+1
        adc     fun’vars+2
        sta     fun’args+1
        jmp     _done

_base:  ldy     #$01
        sty     fun’args
        dey
        sty     fun’args+1

_done:  lda     #$03
        jsr     restore’stack
        rts
.scend
```

The full application, which deals with interfacing with CBM BASIC and handles console I/O and such, is in *fibonacci.oph*.

Appendix A. Example Programs

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

hello1.oph

```
.word $0801
.org  $0801
.outfile "hello.prg"

        .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
```

hello2.oph

```
.word $0801
.org  $0801
.outfile "hello.prg"

.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 "../platform/c64kernal.oph"
```

c64kernal.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
```

hello3.oph

```
.include "c64-1.oph"
.outfile "hello.prg"

.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
```

hello4a.oph

```
.include "c64-1.oph"
.outfile "hello.prg"

.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
        iny
        bne -
        dex
        bne -
        rts
```

hello4b.oph

```
.include "c64-1.oph"
.outfile "hello.prg"

.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
```

hello4c.oph

```
.include "c64-1.oph"
.outfile "hello.prg"

.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
```

hello5.oph

```
.include "c64-1.oph"
.outfile "hello.prg"

.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
```

hello6.oph

```
.include "c64-1.oph"
.outfile "hello.prg"

.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
```

hello7.oph

```
.include "../platform/c64_0.oph"
.require "../platform/c64kernal.oph"
.outfile "hello.prg"

.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
```

structuredemo.oph

```
.include "../platform/c64_0.oph"
.require "../platform/c64kernal.oph"
.outfile "structuredemo.prg"

        jsr print’unsorted
        jsr insertion’sort
        jsr print’list
        rts

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Linked list data: head, next, lb, hb.
; lb/hb: Low/high bytes of the data array.  These are immutable and
;        kept with the program text.
; head:  Array index of the first element in the list, or #$FF if the
;        list is empty
; next:  Array of successor indices.  If you’ve just read element X,
;        the value of memory location next+X is the index of the
;        next element.  If next is #$FF, you’ve reached the end of
;        the list.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

.data
.org    $C000
.space  head    1
.space  next    16

.text
lb:   .byte <$838,<$618,<$205,<$984,<$724,<$301,<$249,<$946
      .byte <$925,<$043,<$114,<$697,<$985,<$633,<$312,<$086
hb:   .byte >$838,>$618,>$205,>$984,>$724,>$301,>$249,>$946
      .byte >$925,>$043,>$114,>$697,>$985,>$633,>$312,>$086

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; insertion’sort:  Sorts the list defined by head, next, hb, lb.
; Arguments:  None.
; Modifies:   All registers destroyed, head and next array sorted.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

insertion’sort:
        lda #$FF        ; Clear list by storing the terminator in ’head’
        sta head
        ldx #$0         ; Loop through the lb/hb array, adding each
insertion’sort’loop:    ; element one at a time
        txa
        pha
        jsr insert_elt
        pla
        tax
        inx
        cpx #$10
        bne insertion’sort’loop
        rts

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; insert_elt: Insert an element into the linked list.  Maintains the
;             list in sorted, ascending order.  Used by
;             insertion’sort.
; Arguments:  X register holds the index of the element to add.
; Modifies:   All registers destroyed; head and next arrays updated
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

.data
.space lbtoinsert 1
.space hbtoinsert 1
```

```
.space indextoinsert 1

.text

insert_elt:
        ldy head                        ; If the list is empty, make
        cpy #$FF                        ; head point at it, and return.
        bne insert_elt’list’not’empty
        stx head
        tya
        sta next,x
        rts
insert_elt’list’not’empty:
        lda lb,x                        ; Cache the data we’re inserting
        sta lbtoinsert
        lda hb,x
        sta hbtoinsert
        stx indextoinsert
        ldy head                        ; Compare the first value with
        sec                             ; the data.  If the data must
        lda lb,y                        ; be inserted at the front...
        sbc lbtoinsert
        lda hb,y
        sbc hbtoinsert
        bmi insert_elt’not’smallest
        tya                             ; Set its next pointer to the
        sta next,x                      ; old head, update the head
        stx head                        ; pointer, and return.
        rts
insert_elt’not’smallest:
        ldx head
insert_elt’loop:                        ; At this point, we know that
        lda next,x                      ; argument > data[X].
        tay
        cpy #$FF                        ; if next[X] = #$FF, insert arg at end.
        beq insert_elt’insert’after’current
        lda lb,y                        ; Otherwise, compare arg to
        sec                             ; data[next[X]].  If we insert
        sbc lbtoinsert                  ; before that...
        lda hb,y
        sbc hbtoinsert
        bmi insert_elt’goto’next
insert_elt’insert’after’current:        ; Fix up all the next links
        tya
        ldy indextoinsert
        sta next,y
        tya
        sta next,x
        rts                             ; and return.
insert_elt’goto’next:                   ; Otherwise, let X = next[X]
        tya                             ; and go looping again.
        tax
        jmp insert_elt’loop

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; print’unsorted: Steps through the data array and prints each value.
; Standalone procedure.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

print’unsorted:
        lda #<unsorted’hdr
        ldx #>unsorted’hdr
        jsr put’string
        ldy #$00
print’unsorted’loop:
        lda hb, Y
        jsr print’hex
        lda lb, y
        jsr print’hex
        lda #$20
        jsr chrout
        iny
        cpy #$10
        bne print’unsorted’loop
        lda #$0D
        jsr chrout
        rts

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; print’list: Starts at head, and prints out every value in the
;             linked list.
; Standalone procedure.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

print’list:
        lda #<sorted’hdr
        ldx #>sorted’hdr
        jsr put’string
        ldy head
print’list’loop:
        cpy #$FF
        beq print’list’done
        lda hb, y
        jsr print’hex
        lda lb, y
        jsr print’hex
        lda #$20
        jsr chrout
        lda next, Y
        tay
        jmp print’list’loop
print’list’done:
        lda #$0d
        jsr chrout
        rts

;; String data for the above routines.

unsorted’hdr:
        .byte 147               ; Clear screen first!
        .byte "UNSORTED DATA:",13,0

sorted’hdr:
        .byte "SORTED DATA:",13,0


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; print’hex: outputs a two-character hex representation of a one-
;            byte value.
; Arguments: Byte to print in accumulator
; Modifies: .A and .X
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

print’hex:
        pha
        clc
        lsr
        lsr
        lsr
        lsr
        tax
        lda hexstr,x
        jsr chrout
        pla
        and #$0F
        tax
        lda hexstr,X
        jsr chrout
        rts

; Character data array for print’hex.
hexstr: .byte "0123456789ABCDEF"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; put’string: outputs a C-style null terminated string with length
;             less than 256 to the screen.  If 256 bytes are written
;             without finding a terminator, the routine ends quietly.
; Arguments: Low byte of string address in .A, high byte in .X
; Modifies: .A and .Y
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

.data zp
.space put’string’addr 2

.text
put’string:
        sta put’string’addr
        stx put’string’addr+1
        ldy #$00
put’string’loop:
        lda (put’string’addr),y
        beq put’string’done
        jsr chrout
        iny
        bne put’string’loop
put’string’done:
        rts
```

fibonacci.oph

```
.include "../platform/c64_0.oph"
.require "../platform/c64kernal.oph"
.outfile "fibonacci.prg"

        lda     #<opening     ; Print opening text
        sta     fun’args
        lda     #>opening
        sta     fun’args+1
        jsr     print’string

        lda     #$00
        sta     fun’vars      ; Count num from 0 to 19
*       lda     fun’vars      ; Main loop: print num, with leading space if <10
        cmp     #$09
        bcs     +
        lda     #$20
        jsr     chrout
        lda     fun’vars
*       sta     fun’args      ; Copy num to args, print it, plus ": "
        inc     fun’args
        lda     #$00
        sta     fun’args+1
        jsr     print’dec
        lda     #$3A
        jsr     chrout
        lda     #$20
        jsr     chrout
        lda     fun’vars      ; Copy num to args, call fib, print result
        sta     fun’args
        jsr     fib
        jsr     print’dec
        lda     #$0D          ; Newline
        jsr     chrout
        inc     fun’vars      ; Increment num; if it’s 20, we’re done.
        lda     fun’vars
        cmp     #20
        bne     --            ; Otherwise, loop.
        rts

opening:
.byte   147, "           FIBONACCI SEQUENCE",13,13,0

.scope
; Uint16 fib (Uint8 x): compute Xth fibonnaci number.
; fib(0) = fib(1) = 1.
; Stack usage: 3.

fib:    lda     #$03
        jsr     save’stack

        lda     fun’vars    ; If x < 2, goto _base.
        cmp     #$02
        bcc     _base

        dec     fun’args    ; Otherwise, call fib(x-1)...
        jsr     fib
        lda     fun’args    ; Copy the result to local variable...
        sta     fun’vars+1
        lda     fun’args+1
        sta     fun’vars+2
        lda     fun’vars    ; Call fib(x-2)...
        sec
        sbc     #$02
        sta     fun’args
        jsr     fib
        clc                 ; And add the old result to it, leaving it
        lda     fun’args    ; in the ’result’ location.
        adc     fun’vars+1
        sta     fun’args
        lda     fun’args+1
        adc     fun’vars+2
        sta     fun’args+1
        jmp     _done       ; and then we’re done.

_base:  ldy     #$01        ; In the base case, just copy 1 to the
        sty     fun’args    ; result.
        dey
        sty     fun’args+1

_done:  lda     #$03
        jsr     restore’stack
        rts
.scend

.scope
; Stack routines: init’stack, save’stack, restore’stack
.data zp
.space _sp      $02
.space _counter $01
.space fun’args $10
.space fun’vars $40
```

```
.text
init’stack:
        lda     #$00
        sta     _sp
        lda     #$A0
        sta     _sp+1
        rts

save’stack:
        sta     _counter
        sec
        lda     _sp
        sbc     _counter
        sta     _sp
        lda     _sp+1
        sbc     #$00
        sta     _sp+1
        ldy     #$00
*       lda     fun’vars, y
        sta     (_sp), y
        lda     fun’args, y
        sta     fun’vars, y
        iny
        dec     _counter
        bne -
        rts

restore’stack:
        pha
        sta     _counter
        ldy     #$00
*       lda     (_sp), y
        sta     fun’vars, y
        iny
        dec     _counter
        bne -
        pla
        clc
        adc     _sp
        sta     _sp
        lda     _sp+1
        adc     #$00
        sta     _sp+1
        rts
.scend


; Utility functions.  print’dec prints an unsigned 16-bit integer.
; It’s ugly and long, mainly because we don’t bother with niceties
; like "division".  print’string prints a zero-terminated string.

.scope
.data
.org    fun’args
        .space  _val            2
        .space  _step           2
        .space  _res            1
        .space  _allowzero      1
.text
print’dec:
        lda     #$00
        sta     _allowzero
        lda     #<10000
        sta     _step
        lda     #>10000
        sta     _step+1
        jsr     repsub’16
        lda     #<1000
        sta     _step
        lda     #>1000
        sta     _step+1
        jsr     repsub’16
        lda     #0
        sta     _step+1
        lda     #100
        sta     _step
        jsr     repsub’16
        lda     #10
        sta     _step
        jsr     repsub’16
        lda     _val
        jsr     _print
        rts

repsub’16:
        lda     #$00
        sta     _res
*       lda     _val
        sec
        sbc     _step
        lda     _val+1
        sbc     _step+1
        bcc     _done
        lda     _val
        sec
        sbc     _step
        sta     _val
        lda     _val+1
        sbc     _step+1
        sta     _val+1
        inc     _res
        jmp     -
_done:  lda     _res
        ora     _allowzero
        beq     _ret
        sta     _allowzero
        lda     _res
_print: clc
        adc     #’0
        jsr     chrout
_ret:   rts
.scend

print’string:
        ldy     #$00
*       lda     (fun’args), y
        beq     +
        jsr     chrout
        iny
        jmp     -
*       rts
```

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.

- `.outfile` *filename*: Sets the filename for the output binary if one has not already been set. If no name is ever set, the output will be written to `ophis.bin`.
- `.advance` *address* [*, filler*]: Forces the program counter to be *address*. Unlike the `.org` directive, `.advance` outputs bytes (the value of *filler*, or zeroes if it is unspecified) 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.
- `.cbmfloat` *string* [, *string*, ...]: Specifies a series of strings, which are interpreted as floating point constants, and then included in the 5-byte floating point format used by the Commodore BASICs. This format is 8 bits of exponent, followed by a sign bit and a 31-bit big-endian mantissa fraction. (The 1 in front of the binary point is presumed to be present.) An exponent of 0 specifies a constant of 0, and the exponent is shifted up by 129 before being stored.

  Because IEEE-754 doesn't perfectly match the Commodore's system, if you wish to precisely replicate individual constants that cannot be represented exactly you may have better luck with the following program, which will run on both the Commodore 64 and VIC-20:

  ```
  10 CLR:V=0:PV=PEEK(45)+256*PEEK(46)+2
  20 INPUT "NUMBER (0 TO QUIT)";V
  30 IF V=0 THEN END
  40 PRINT ".BYTE";
  50 FOR I=0 TO 4
  60 IF I>0 THEN PRINT CHR$(157);",";
  70 PRINT PEEK(PV+I);:NEXT I:PRINT:GOTO 20
  ```

This program will print out a `.byte` directive for you to include in your program to represent that number.

- `.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* [, *offset* [, *length*]]: Inserts the contents of the file specified as binary data. Use it to include graphics information, precompiled code, or other non-assembler data. You may also optionally specify an index to start including from, or a length to only include a subset.
- `.include` *filename*: Includes the entirety of the file specified at that point in the program. Use this to order your final sources, if you aren't doing it via the command line.
- `.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.