Introduction

In this tiny ebook I'm going to show you how to get started writing 6502 assembly language. The 6502 processor was massive in the seventies and eighties, powering famous computers like the BBC Micro, Atari 2600, Commodore 64, Apple II, and the Nintendo Entertainment System. Bender in Futurama has a 6502 processor for a brain. Even the Terminator was programmed in 6502.

So, why would you want to learn 6502? It's a dead language isn't it? Well, so's Latin. And they still teach that. Q.E.D.

(Actually, I've been reliably informed that 6502 processors are still being produced by Western Design Center, so clearly 6502 *isn't* a dead language! Who knew?)

Seriously though, I think it's valuable to have an understanding of assembly language. Assembly language is the lowest level of abstraction in computers - the point at which the code is still readable. Assembly language translates directly to the bytes that are executed by your computer's processor. If you understand how it works, you've basically become a computer magician.

Then why 6502? Why not a *useful* assembly language, like x86? Well, I don't think learning x86 is useful. I don't think you'll ever have to *write* assembly language in your day job - this is purely an academic exercise, something to expand your mind and your thinking. 6502 was originally written in a different age, a time when the majority of developers were writing assembly directly, rather than in these new-fangled high-level programming languages. So, it was designed to be written by humans. More modern assembly languages are meant to written by compilers, so let's leave it to them. Plus, 6502 is *fun*. Nobody ever called x86 *fun*.

Our first program

So, let's dive in! That thing below is a little JavaScript 6502 assembler and simulator that I adapted for this book. Click **Assemble** then **Run** to assemble and run the snippet of assembly language.

Hopefully the black area on the right now has three coloured "pixels" at the top left. (If this doesn't work, you'll probably need to upgrade your browser to something more modern, like Chrome or Firefox.)

So, what's this program actually doing? Let's step through it with the debugger. Hit **Reset**, then check the **Debugger** checkbox to start the debugger. Click **Step** once. If you were watching carefully, you'll have noticed that `A=` changed from `$00` to `$01`, and `PC=` changed from `$0600` to `$0602`.

Any numbers prefixed with `$` in 6502 assembly language (and by extension, in this book) are in hexadecimal (hex) format. If you're not familiar with hex numbers, I recommend you read the Wikipedia article. Anything prefixed with `#` is a literal number value. Any other number refers to a

memory location.

Equipped with that knowledge, you should be able to see that the instruction `LDA #$01` loads the hex value `$01` into register `A`. I'll go into more detail on registers in the next section.

Press **Step** again to execute the second instruction. The top-left pixel of the simulator display should now be white. This simulator uses the memory locations `$0200 to $05ff` to draw pixels on its display. The values `$00` to `$0f` represent 16 different colours (`$00` is black and `$01` is white), so storing the value `$01` at memory location `$0200` draws a white pixel at the top left corner. This is simpler than how an actual computer would output video, but it'll do for now.

So, the instruction `STA $0200` stores the value of the `A` register to memory location `$0200`. Click **Step** four more times to execute the rest of the instructions, keeping an eye on the `A` register as it changes.

Exercises

1. Try changing the colour of the three pixels.
2. Change one of the pixels to draw at the bottom-right corner (memory location `$05ff`).
3. Add more instructions to draw extra pixels.

Registers and flags

We've already had a little look at the processor status section (the bit with `A`, `PC` etc.), but what does it all mean?

The first line shows the `A`, `X` and `Y` registers (`A` is often called the "accumulator"). Each register holds a single byte. Most operations work on the contents of these registers.

`SP` is the stack pointer. I won't get into the stack yet, but basically this register is decremented every time a byte is pushed onto the stack, and incremented when a byte is popped off the stack.

`PC` is the program counter - it's how the processor knows at what point in the program it currently is. It's like the current line number of an executing script. In the JavaScript simulator the code is assembled starting at memory location `$0600`, so `PC` always starts there.

The last section shows the processor flags. Each flag is one bit, so all seven flags live in a single byte. The flags are set by the processor to give information about the previous instruction. More on that later. Read more about the registers and flags here.

Instructions

Instructions in assembly language are like a small set of predefined functions. All instructions take zero or one arguments. Here's some annotated source code to introduce a few different instructions:

Assemble the code, then turn on the debugger and step through the code, watching the `A` and `X` registers. Something slightly odd happens on the line `ADC #$c4`. You might expect that adding `$c4` to `$c0` would give `$184`, but this processor gives the result as `$84`. What's up with that?

The problem is, `$184` is too big to fit in a single byte (the max is `$FF`), and the registers can only hold a single byte. It's OK though; the processor isn't actually dumb. If you were looking carefully enough, you'll have noticed that the carry flag was set to `1` after this operation. So that's how you know.

In the simulator below **type** (don't paste) the following code:

```
LDA #$80
STA $01
ADC $01
```

An important thing to notice here is the distinction between `ADC #$01` and `ADC $01`. The first one adds the value `$01` to the `A` register, but the second adds the value stored at memory location `$01` to the `A` register.

Assemble, check the **Monitor** checkbox, then step through these three instructions. The monitor shows a section of memory, and can be helpful to visualise the execution of programs. `STA $01` stores the value of the `A` register at memory location `$01`, and `ADC $01` adds the value stored at the memory location `$01` to the `A` register. `$80 + $80` should equal `$100`, but because this is bigger than a byte, the `A` register is set to `$00` and the carry flag is set. As well as this though, the zero flag is set. The zero flag is set by all instructions where the result is zero.

A full list of the 6502 instruction set is available here and here (I usually refer to both pages as they have their strengths and weaknesses). These pages detail the arguments to each instruction, which registers they use, and which flags they set. They are your bible.

Exercises

1. You've seen `TAX`. You can probably guess what `TAY`, `TXA` and `TYA` do, but write some code to test your assumptions.
2. Rewrite the first example in this section to use the `Y` register instead of the `X` register.
3. The opposite of `ADC` is `SBC` (subtract with carry). Write a program that uses this instruction.

Branching

So far we're only able to write basic programs without any branching logic. Let's change that.

6502 assembly language has a bunch of branching instructions, all of which branch based on whether certain flags are set or not. In this example we'll be looking at `BNE`: "Branch on not equal".

First we load the value `$08` into the `X` register. The next line is a label. Labels just mark certain points in a program so we can return to them later. After the label we decrement `X`, store it to `$0200` (the top-left pixel), and then compare it to the value `$03`. `CPX` compares the value in the `X` register with another value. If the two values are equal, the `Z` flag is set to `1`, otherwise it is set to `0`.

The next line, `BNE decrement`, will shift execution to the decrement label if the `Z` flag is set to `0` (meaning that the two values in the `CPX` comparison were not equal), otherwise it does nothing and we store `X` to `$0201`, then finish the program.

In assembly language, you'll usually use labels with branch instructions. When assembled though, this label is converted to a single-byte relative offset (a number of bytes to go backwards or forwards from the next instruction) so branch instructions can only go forward and back around 256 bytes. This means they can only be used to move around local code. For moving further you'll need to use the jumping instructions.

Exercises

1. The opposite of `BNE` is `BEQ`. Try writing a program that uses `BEQ`.
2. `BCC` and `BCS` ("branch on carry clear" and "branch on carry set") are used to branch on the carry flag. Write a program that uses one of these two.

Addressing modes

The 6502 uses a 16-bit address bus, meaning that there are 65536 bytes of memory available to the processor. Remember that a byte is represented by two hex characters, so the memory locations are generally represented as `$0000 – $ffff`. There are various ways to refer to these memory locations, as detailed below.

With all these examples you might find it helpful to use the memory monitor to watch the memory change. The monitor takes a starting memory location and a number of bytes to display from that location. Both of these are hex values. For example, to display 16 bytes of memory from `$c000`, enter `c000` and `10` into **Start** and **Length**, respectively.

Absolute: `$c000`

With absolute addressing, the full memory location is used as the argument to the instruction. For example:

```
STA $c000 ;Store the value in the accumulator at memory location $c000
```

Zero page: `$c0`

All instructions that support absolute addressing (with the exception of the jump instructions) also

have the option to take a single-byte address. This type of addressing is called “zero page” - only the first page (the first 256 bytes) of memory is accessible. This is faster, as only one byte needs to be looked up, and takes up less space in the assembled code as well.

Zero page,X: `$c0,x`

This is where addressing gets interesting. In this mode, a zero page address is given, and then the value of the `X` register is added. Here is an example:

```
LDX #$01   ;X is $01
LDA #$aa   ;A is $aa
STA $a0,X ;Store the value of A at memory location $a1
INX        ;Increment X
STA $a0,X ;Store the value of A at memory location $a2
```

If the result of the addition is larger than a single byte, the address wraps around. For example:

```
LDX #$05
STA $ff,X ;Store the value of A at memory location $04
```

Zero page,Y: `$c0,y`

This is the equivalent of zero page,X, but can only be used with `LDX` and `STX`.

Absolute,X and absolute,Y: `$c000,x` and `$c000,y`

These are the absolute addressing versions of zero page,X and zero page,Y. For example:

```
LDX #$01
STA $0200,X ;Store the value of A at memory location $0201
```

Immediate: `#$c0`

Immediate addressing doesn’t strictly deal with memory addresses - this is the mode where actual values are used. For example, `LDX #$01` loads the value `$01` into the `X` register. This is very different to the zero page instruction `LDX $01` which loads the value at memory location `$01` into the `X` register.

Relative: `$c0` (or label)

Relative addressing is used for branching instructions. These instructions take a single byte, which is used as an offset from the following instruction.

Assemble the following code, then click the **Hexdump** button to see the assembled code.

The hex should look something like this:

```
a9 01 c9 02 d0 02 85 22 00
```

`a9` and `c9` are the processor opcodes for immediate-addressed `LDA` and `CMP` respectively. `01` and `02` are the arguments to these instructions. `d0` is the opcode for `BNE`, and its argument is `02`. This means "skip over the next two bytes" (`85 22`, the assembled version of `STA $22`). Try editing the code so `STA` takes a two-byte absolute address rather than a single-byte zero page address (e.g. change `STA $22` to `STA $2222`). Reassemble the code and look at the hexdump again - the argument to `BNE` should now be `03`, because the instruction the processor is skipping past is now three bytes long.

Implicit

Some instructions don't deal with memory locations (e.g. `INX` - increment the `X` register). These are said to have implicit addressing - the argument is implied by the instruction.

Indirect: `($c000)`

Indirect addressing uses an absolute address to look up another address. The first address gives the least significant byte of the address, and the following byte gives the most significant byte. That can be hard to wrap your head around, so here's an example:

In this example, `$f0` contains the value `$01` and `$f1` contains the value `$cc`. The instruction `JMP ($f0)` causes the processor to look up the two bytes at `$f0` and `$f1` (`$01` and `$cc`) and put them together to form the address `$cc01`, which becomes the new program counter. Assemble and step through the program above to see what happens. I'll talk more about `JMP` in the section on Jumping.

Indexed indirect: `($c0,X)`

This one's kinda weird. It's like a cross between zero page,X and indirect. Basically, you take the zero page address, add the value of the `X` register to it, then use that to look up a two-byte address. For example:

Memory locations `$01` and `$02` contain the values `$05` and `$06` respectively. Think of `($00,X)` as `($00 + X)`. In this case `X` is `$01`, so this simplifies to `($01)`. From here things proceed like standard indirect addressing - the two bytes at `$01` and `$02` (`$05` and `$06`) are looked up to form the address `$0605`. This is the address that the `Y` register was stored into in the previous instruction, so the `A` register gets the same value as `Y`, albeit through a much more circuitous route. You won't see this much.

Indirect indexed: `($c0),Y`

Indirect indexed is like indexed indirect but less insane. Instead of adding the `X` register to the address *before* dereferencing, the zero page address is dereferenced, and the `Y` register is added to the resulting address.

In this case, `($01)` looks up the two bytes at `$01` and `$02`: `$03` and `$07`. These form the address `$0703`. The value of the `Y` register is added to this address to give the final address `$0704`.

Exercise

1. Try to write code snippets that use each of the 6502 addressing modes. Remember, you can use the monitor to watch a section of memory.

The stack

The stack in a 6502 processor is just like any other stack - values are pushed onto it and popped ("pulled" in 6502 parlance) off it. The current depth of the stack is measured by the stack pointer, a special register. The stack lives in memory between `$0100` and `$01ff`. The stack pointer is initially `$ff`, which points to memory location `$01ff`. When a byte is pushed onto the stack, the stack pointer becomes `$fe`, or memory location `$01fe`, and so on.

Two of the stack instructions are `PHA` and `PLA`, "push accumulator" and "pull accumulator". Below is an example of these two in action.

`X` holds the pixel colour, and `Y` holds the position of the current pixel. The first loop draws the current colour as a pixel (via the `A` register), pushes the colour to the stack, then increments the colour and position. The second loop pops the stack, draws the popped colour as a pixel, then increments the position. As should be expected, this creates a mirrored pattern.

Jumping

Jumping is like branching with two main differences. First, jumps are not conditionally executed, and second, they take a two-byte absolute address. For small programs, this second detail isn't very important, as you'll mostly be using labels, and the assembler works out the correct memory location from the label. For larger programs though, jumping is the only way to move from one section of the code to another.

JMP

`JMP` is an unconditional jump. Here's a really simple example to show it in action:

JSR/RTS

`JSR` and `RTS` ("jump to subroutine" and "return from subroutine") are a dynamic duo that you'll

usually see used together. `JSR` is used to jump from the current location to another part of the code. `RTS` returns to the previous position. This is basically like calling a function and returning.

The processor knows where to return to because `JSR` pushes the address minus one of the next instruction onto the stack before jumping to the given location. `RTS` pops this location, adds one to it, and jumps to that location. An example:

The first instruction causes execution to jump to the `init` label. This sets `X`, then returns to the next instruction, `JSR loop`. This jumps to the `loop` label, which increments `X` until it is equal to `$05`. After that we return to the next instruction, `JSR end`, which jumps to the end of the file. This illustrates how `JSR` and `RTS` can be used together to create modular code.

Creating a game

Now, let's put all this knowledge to good use, and make a game! We're going to be making a really simple version of the classic game 'Snake'.

Even though this will be a simple version, the code will be substantially larger than all the previous examples. We will need to keep track of several memory locations together for the various aspects of the game. We can still do the necessary bookkeeping throughout the program ourselves, as before, but on a larger scale that quickly becomes tedious and can also lead to bugs that are difficult to spot. Instead we'll now let the assembler do some of the mundane work for us.

In this assembler, we can define descriptive constants (or symbols) that represent numbers. The rest of the code can then simply use the constants instead of the literal number, which immediately makes it obvious what we're dealing with. You can use letters, digits and underscores in a name.

Here's an example. Note that immediate operands are still prefixed with a #.

The simulator widget below contains the entire source code of the game. I'll explain how it works in the following sections.

Willem van der Jagt made a fully annotated gist of this source code, so follow along with that for more details.

Overall structure

After the initial block of comments (lines starting with semicolons), the first two lines are:

```
jsr init
jsr loop
```

`init` and `loop` are both subroutines. `init` initializes the game state, and `loop` is the main game loop.

The `loop` subroutine itself just calls a number of subroutines sequentially, before looping back on itself:

```
loop:
  jsr readkeys
  jsr checkCollision
  jsr updateSnake
  jsr drawApple
  jsr drawSnake
  jsr spinwheels
  jmp loop
```

First, `readkeys` checks to see if one of the direction keys (W, A, S, D) was pressed, and if so, sets the direction of the snake accordingly. Then, `checkCollision` checks to see if the snake collided with itself or the apple. `updateSnake` updates the internal representation of the snake, based on its direction. Next, the apple and snake are drawn. Finally, `spinWheels` makes the processor do some busy work, to stop the game from running too quickly. Think of it like a sleep command. The game keeps running until the snake collides with the wall or itself.

Zero page usage

The zero page of memory is used to store a number of game state variables, as noted in the comment block at the top of the game. Everything in `$00`, `$01` and `$10` upwards is a pair of bytes representing a two-byte memory location that will be looked up using indirect addressing. These memory locations will all be between `$0200` and `$05ff` - the section of memory corresponding to the simulator display. For example, if `$00` and `$01` contained the values `$01` and `$02`, they would be referring to the second pixel of the display (`$0201` - remember, the least significant byte comes first in indirect addressing).

The first two bytes hold the location of the apple. This is updated every time the snake eats the apple. Byte `$02` contains the current direction. `1` means up, `2` right, `4` down, and `8` left. The reasoning behind these numbers will become clear later.

Finally, byte `$03` contains the current length of the snake, in terms of bytes in memory (so a length of `4` means `2` pixels).

Initialization

The `init` subroutine defers to two subroutines, `initSnake` and `generateApplePosition`. `initSnake` sets the snake direction, length, and then loads the initial memory locations of the snake head and body. The byte pair at `$10` contains the screen location of the head, the pair at `$12` contains the location of the single body segment, and `$14` contains the location of the tail (the tail is the last

segment of the body and is drawn in black to keep the snake moving). This happens in the following code:

```
lda #$11
sta $10
lda #$10
sta $12
lda #$0f
sta $14
lda #$04
sta $11
sta $13
sta $15
```

This loads the value `$11` into the memory location `$10`, the value `$10` into `$12`, and `$0f` into `$14`. It then loads the value `$04` into `$11`, `$13` and `$15`. This leads to memory like this:

```
0010: 11 04 10 04 0f 04
```

which represents the indirectly-addressed memory locations `$0411`, `$0410` and `$040f` (three pixels in the middle of the display). I'm labouring this point, but it's important to fully grok how indirect addressing works.

The next subroutine, `generateApplePosition`, sets the apple location to a random position on the display. First, it loads a random byte into the accumulator (`$fe` is a random number generator in this simulator). This is stored into `$00`. Next, a different random byte is loaded into the accumulator, which is then `AND`-ed with the value `$03`. This part requires a bit of a detour.

The hex value `$03` is represented in binary as `00000111`. The `AND` opcode performs a bitwise AND of the argument with the accumulator. For example, if the accumulator contains the binary value `01010101`, then the result of `AND` with `00000111` will be `00000101`.

The effect of this is to mask out the least significant three bytes of the accumulator, setting the others to zero. This converts a number in the range of 0–255 to a number in the range of 0–3.

After this, the value 2 is added to the accumulator, to create a final random number in the range 2–5.

The result of this subroutine is to load a random byte into `$00`, and a random number between 2 and 5 into `$01`. Because the least significant byte comes first with indirect addressing, this translates into a memory address between `$0200` and `$05ff`: the exact range used to draw the display.

The game loop

Nearly all games have at their heart a game loop. All game loops have the same basic form: accept user input, update the game state, and render the game state. This loop is no different.

Reading the input

The first subroutine, `readKeys`, takes the job of accepting user input. The memory location `$ff` holds the ascii code of the most recent key press in this simulator. The value is loaded into the accumulator, then compared to `$77` (the hex code for W), `$64` (D), `$73` (S) and `$61`. If any of these comparisons are successful, the program branches to the appropriate section. Each section (`upKey`, `rightKey`, etc.) first checks to see if the current direction is the opposite of the new direction. This requires another little detour.

As stated before, the four directions are represented internally by the numbers 1, 2, 4 and 8. Each of these numbers is a power of 2, thus they are represented by a binary number with a single 1:

```
1 => 0001 (up)
2 => 0010 (right)
4 => 0100 (down)
8 => 1000 (left)
```

The `BIT` opcode is similar to `AND`, but the calculation is only used to set the zero flag - the actual result is discarded. The zero flag is set only if the result of AND-ing the accumulator with argument is zero. When we're looking at powers of two, the zero flag will only be set if the two numbers are not the same. For example, `0001 AND 0001` is not zero, but `0001 AND 0010` is zero.

So, looking at `upKey`, if the current direction is down (4), the bit test will be zero. `BNE` means "branch if the zero flag is clear", so in this case we'll branch to `illegalMove`, which just returns from the subroutine. Otherwise, the new direction (1 in this case) is stored in the appropriate memory location.

Updating the game state

The next subroutine, `checkCollision`, defers to `checkAppleCollision` and `checkSnakeCollision`. `checkAppleCollision` just checks to see if the two bytes holding the location of the apple match the two bytes holding the location of the head. If they do, the length is increased and a new apple position is generated.

`checkSnakeCollision` loops through the snake's body segments, checking each byte pair against the head pair. If there is a match, then game over.

After collision detection, we update the snake's location. This is done at a high level like so: First, move each byte pair of the body up one position in memory. Second, update the head according to the current direction. Finally, if the head is out of bounds, handle it as a collision. I'll illustrate this with

some ascii art. Each pair of brackets contains an x,y coordinate rather than a pair of bytes for simplicity.

```
  0     1     2     3     4
Head                    Tail

[1,5][1,4][1,3][1,2][2,2]    Starting position

[1,5][1,4][1,3][1,2][1,2]    Value of (3) is copied into (4)

[1,5][1,4][1,3][1,3][1,2]    Value of (2) is copied into (3)

[1,5][1,4][1,4][1,3][1,2]    Value of (1) is copied into (2)

[1,5][1,5][1,4][1,3][1,2]    Value of (0) is copied into (1)

[0,5][1,5][1,4][1,3][1,2]    Value of (0) is updated based on direction
```

At a low level, this subroutine is slightly more complex. First, the length is loaded into the `X` register, which is then decremented. The snippet below shows the starting memory for the snake.

```
Memory location: $10 $11 $12 $13 $14 $15

Value:           $11 $04 $10 $04 $0f $04
```

The length is initialized to `4`, so `X` starts off as `3`. `LDA $10,x` loads the value of `$13` into `A`, then `STA $12,x` stores this value into `$15`. `X` is decremented, and we loop. Now `X` is `2`, so we load `$12` and store it into `$14`. This loops while `X` is positive (`BPL` means “branch if positive”).

Once the values have been shifted down the snake, we have to work out what to do with the head. The direction is first loaded into `A`. `LSR` means “logical shift right”, or “shift all the bits one position to the right”. The least significant bit is shifted into the carry flag, so if the accumulator is `1`, after `LSR` it is `0`, with the carry flag set.

To test whether the direction is `1`, `2`, `4` or `8`, the code continually shifts right until the carry is set. One `LSR` means “up”, two means “right”, and so on.

The next bit updates the head of the snake depending on the direction. This is probably the most complicated part of the code, and it’s all reliant on how memory locations map to the screen, so let’s look at that in more detail.

You can think of the screen as four horizontal strips of 32 × 8 pixels. These strips map to `$0200–$02ff`, `$0300–$03ff`, `$0400–$04ff` and `$0500–$05ff`. The first rows of pixels are `$0200–$021f`, `$0220–$023f`, `$0240–$025f`, etc.

As long as you're moving within one of these horizontal strips, things are simple. For example, to move right, just incrememnt the least significant byte (e.g. `$0200` becomes `$0201`). To go down, add `$20` (e.g. `$0200` becomes `$0220`). Left and up are the reverse.

Going between sections is more complicated, as we have to take into account the most significant byte as well. For example, going down from `$02e1` should lead to `$0301`. Luckily, this is fairly easy to accomplish. Adding `$20` to `$e1` results in `$01` and sets the carry bit. If the carry bit was set, we know we also need to increment the most significant byte.

After a move in each direction, we also need to check to see if the head would become out of bounds. This is handled differently for each direction. For left and right, we can check to see if the head has effectively "wrapped around". Going right from `$021f` by incrementing the least significant byte would lead to `$0220`, but this is actually jumping from the last pixel of the first row to the first pixel of the second row. So, every time we move right, we need to check if the new least significant byte is a multiple of `$20`. This is done using a bit check against the mask `$1f`. Hopefully the illustration below will show you how masking out the lowest 5 bits reveals whether a number is a multiple of `$20` or not.

```
$20: 0010 0000
$40: 0100 0000
$60: 0110 0000

$1f: 0001 1111
```

I won't explain in depth how each of the directions work, but the above explanation should give you enough to work it out with a bit of study.

Rendering the game

Because the game state is stored in terms of pixel locations, rendering the game is very straightforward. The first subroutine, `drawApple`, is extremely simple. It sets `Y` to zero, loads a random colour into the accumulator, then stores this value into `($00),y`. `$00` is where the location of the apple is stored, so `($00),y` dereferences to this memory location. Read the "Indirect indexed" section in Addressing modes for more details.

Next comes `drawSnake`. This is pretty simple too. `X` is set to zero and `A` to one. We then store `A` at `($10,x)`. `$10` stores the two-byte location of the head, so this draws a white pixel at the current head position. Next we load `$03` into `X`. `$03` holds the length of the snake, so `($10,x)` in this case will be

the location of the tail. Because `A` is zero now, this draws a black pixel over the tail. As only the head and the tail of the snake move, this is enough to keep the snake moving.

The last subroutine, `spinWheels`, is just there because the game would run too fast otherwise. All `spinWheels` does is count `X` down from zero until it hits zero again. The first `dex` wraps, making `X` `#$ff`.