Subject: Writing subroutines using parameters in MOS 6502 assembly
Author: Wouter Bovelander
Date: june 14th, 2015

When writing computer programs it is wise to use some form of structure to prevent code from becoming hard or impossible to maintain. Best practices are described in the field of Structured Programming, a programming paradigm aimed at using subroutines (among others) to prevent your code from becoming a mess. A subroutine is a callable piece of code like a function or a method to allow a sequence to be used as a single statement. More often than not parameters are passed to the subroutine to influence its outcome.

Writing subroutines in 6502 assembly is apparently easy. The 6502 has two operations to support subroutines that do everything for you: JSR and RTS. A little example:

```
;main code
           LDA #43
           JSR add-five
           TAX
           …
           …

;subroutine add-five
add-five   ADD #5
           RTS
```

The example shows a main route loading a value into the accumulator and calling a subroutine called ‘add-five’, which predictably adds five to the accumulator. The routine is called, five is added and the RTS operation is executed sending us back to the next instruction (TAX in this case). This immediately shows a simple form of using parameters: the accumulator is used to pass the initial value to the subroutine. It is used again to pass the changed value back to the main program. However, this is not a very flexible way. The registers are a valuable bunch and are probably being used for other things than to pass parameters.

## THE STACK

The stack is a piece of memory located at locations $0100 to $01FF. In an empty stack the stack pointer (a single byte register) has the value $FF, meaning that as the stack grows the stack pointer is decreased. Remember that. It’s important. It’s also important to realise that the stack pointer points to the first empty location on the stack.

What you cannot see is how the stack is used to perform the operation above. The 6502 has as instruction pointer (IP) which always points to the memory location holding the next instruction to be executed. Each memory location can be described using two bytes: a low byte and a high byte. At the moment the JSR operation is called the 6502 first stores the address of the next operation. It uses the stack to do that. In our case the address of the TAX operation is stored onto the stack. Next the IP is updated to the address being called in the JSR operation. In our case the address of the add-five subroutine. At the end of the subroutine the RTS operation is executed. This operation pops two bytes off the stack, stuffs the values in the IP and lets everything from there. Whatever is on the stack at that moment.

> **Aside**. In reality the two bytes of the next address are added to the stack and 1 is subtracted from that address. The RTS operation pops the address and adds 1. As a programmer you don’t need to know, but it’s better that you are completely informed.

## PARAMETERS

There are several ways of passing parameters to our subroutine. We could dedicate a fixed place in memory to store values in that could be reached by both our main program and our subroutine. But what if our subroutine called another subroutine. We would need other memory locations so not as to overwrite our first ones. But what if there was another subroutine? Forget it..nested subroutines kills all the fun. Another tricky thing here is that these memory locations would be like global variables. They are hell to debug, so let's think of something else.
What if we reserved memory locations inside the subroutine itself? That way we would be creating local variables. Yeah, that could work. As long as we don't think of all the unused memory we're wasting. Sure in 2015 that is of no real consequence but back in the 1980's memory was expensive and a rare commodity. It's plausible but not so efficient.
So we need a piece of memory that is accessible by all of our program, is temporary in use so we won't be creating the global variable effect and is efficient. How about the stack?

PARAMETERS AND THE STACK

Now, this idea of the stack is enticing because we can use it to put parameters on it to pass to the subroutine. However, we now know that the 6502 is also messing about with the stack. It serves two purposes so we must be careful not to leave the stack with the wrong information because we know the IP is going to be loaded with what is on the stack regardless of what it is. We may send the 6502 into a spin from which it can never recover.

So have a look at this:

```
;main code
           LDA #43
           PHA
           JSR add-five
return     PLA
           TAX
           …
           …

;subroutine add-five
add-five   PLA
           ADD #5
           PHA
           RTS
```

In this example we load #43 into the accumulator and push it onto the stack. Next we call the subroutine which pulls it off the stack and into the accumulator, adds five and pushes it back onto the stack and then returns to the main code. In the main code we simply pull the first value off the stack into the accumulator and presto. Right? Wrong. We forgot about the 6502's stack handling.

Ok first of all we need to realise that after we've pushed our #43 onto the stack, two more bytes were added by the JSR command. When we arrive at our add-five subroutine the stack looks a bit like this:

```
SP —>$FC    (EMPTY)
       $FD    LOW_BYTE OF RETURN
       $FE    HIGH_BYTE OF RETURN
       $FF    #43
```

So how do we reach our parameter. We know that SP+1 and SP+2 hold the return address. Our data is to be found at SP+3. Do we pop the first two bytes and store them? Or do we use the SP and index from there? If we can leave the SP alone we can be sure that any RTS will always work

properly. So maybe that's the safest way to go. Moreover we won't need to waste memory to store the return address.

Consider this code:

```
;main code
           LDA #43
           PHA
           JSR add-five
return     PLA
           TAX
           …
           …

;subroutine add-five
add-five   TSX
           INX
           INX
           INX
           LDA $0100,X
           ADD #5
           STA $0100,X
           RTS
```

This code starts by loading a value into the accumulator and pushing it onto the stack. Next the 6502 pushes the two return address bytes onto the stack and jumps to the add-five subroutine. There, the current value of the stack pointer is transferred to the X register. The X register is then increased by three in order to point to the location in the stack where our parameter is. Note that we're not actually doing anything with the stack. We're leaving it in tact. The operation:

LDA $0100,X

Loads the value of the memory address $0100 PLUS the X register into the accumulator. We stored our value of #43 there. Five is added and stored back into the same location. We've basically found a way of messing about in the stack without actually popping or pushing anything. Now, when the RTS is executed the 6502 does it normal thing: pop the return address off the stack and return to the next operation in our main program. Because the parameter value is now the first value on the stack, we can just pop it off and presto, we're done!