Coffee Into Bugs: A 6502 ABI

Dave Allison

*“A software engineer is a device for converting coffee into bugs”*

I’m writing another C compiler that targets the 6502 processor[[1]](#footnote-1) and I need an Application Binary Interface (ABI). There is no defined ABI for the 6502 as far as I can tell, so I made one up. It was fun, so I’ll describe it here just in case anyone is interested.

When I wrote my first C compiler back in 1986 it was for the BBC Micro and was available to purchase as Beebug C. At that time, C compilers for 6502 machines were supplied on multiple floppy disks that you had to juggle to run the various compiler phases. My compiler, linker and interpreter were written entirely in assembly language and fitted into two 16K ROMs. Yes, there was an interpreter. The compiler produced bytecode that I made up, and the interpreter ran that bytecode to execute the program. The result was small object files that ran basically the same speed as the built-in BASIC interpreter.

Now I’m trying again, but this time I want my compiler to produce reasonably fast native machine code instead of an interpreted bytecode. You know, it turns out to be very difficult to generate good quality 6502 machine code that doesn’t completely blow through all your addressable memory space.

You can download a 6502 compiler today and run code on your 6502 machine (that I assume you’ve built for fun because it’s not much use as a real computer). However, if you look at its output, it’s a threaded interpreter – stack based with bytecode instructions replaced by calls to subroutines. While it’s probably faster to execute than a fully interpreted program, it’s still interpreted, with its bytecode instructions encoded as 3-byte JSR instructions.

Here’s what I want. For this code:

#include <stdio.h>

void foo(int a, int b) {

printf("hello world %d\n", a+b);

}

int main() {

foo(1, 2);

}

I want this kind of output[[2]](#footnote-2):

.global foo

.type foo, @function

foo:

ldx #7

jsr \_\_enter\_nomask

ldx #0

jsr \_\_arg\_value2\_i0 // a

ldx #2

jsr \_\_arg\_value2\_i1 // b

clc

lda \_\_i0

adc \_\_i1

sta \_\_i2

lda \_\_i0+1

adc \_\_i1+1

sta \_\_i2+1

jsr \_\_pushi2

ldx #%lo(.str.1)

ldy #%hi(.str.1)

jsr \_\_pushxy

ldx #\_\_i2

ldy #0

jsr printf

jsr \_\_incsp4

ldy #10

jmp \_\_leave\_void\_nomask

.func\_end\_foo:

.size foo, .func\_end\_foo-foo

.global main

.type main, @function

main:

stx \_\_result

sty \_\_result+1

ldx #7

jsr \_\_enter\_nomask

ldx #2

jsr \_\_pushxy0

ldx #1

jsr \_\_pushxy0

jsr foo

jsr \_\_incsp4

stz \_\_i0

stz \_\_i0+1

ldx #8

jsr \_\_load\_result

lda #\_\_i0

jsr \_\_result2

ldy #10

jmp \_\_leave\_nomask

.func\_end\_main:

.size main, .func\_end\_main-main

.data

.section ".rodata", "aMS", @progbits

.str.1:

.asciz "hello world %d\n"

.type .str.1, @object

.size .str.1, 16

The assembler syntax follows the modern style of GAS[[3]](#footnote-3) and I’ve written an assembler to produce ELF files.

The ABI has the following characteristics.

1. Function calls are made using a JSR instruction
2. Arithmetic operations can be done inline (like the sequence to add **a** and **b** in **foo**)
3. Zero-page memory usage for operations with a runtime stack for stack frames.
4. No artificial limit on stack frame size.
5. Support for IEEE754 single precision floating point.
6. Separate compilation with static linking.

# ABI Overview

The 6502 instruction set is a terrible compiler target. You only have 3 8-bit CPU registers to work with, so that means byte-at-a-time operations. Memory addressing is very limited, allowing 8-bit unsigned offsets from a base address only. The way traditional compilers that targeted 6502 would work is to run the code using a runtime stack that holds all the operands for operations. For example, a traditional 16-bit addition would be generated something like this:

push arg1

push arg2

add

pop dest

The runtime stack contains the operands for all operations, which uses the values on the stack and push the result back onto the stack (after popping the argument off). This is a fine way to generate code and is certainly the easiest but isn’t very efficient. You can’t use the 256-byte 6502 CPU stack for the runtime stack as it’s way too small, so each push operation loads the current stack point from two zero-page locations, subtracts 2 from it and copies from another memory location into the address contained in the stack pointer. The *add* operation needs to copy from the stack memory to zero page, while incrementing the stack pointer, perform the operation and then push the result back onto the stack.

If you are writing code in 6502 assembly language you choose a set of zero-page locations for your variables and use some buffers or scratch space at fixed addresses in memory. You can write very efficient code that way, but you are only able to do it because you are deciding the layout and memory usage at the lowest level. My goal for this ABI is to get as close to hand-written assembler memory usage as I can.

A modern CPU, like the ARMv8 or RISC-V has a nice large set of 64-bit registers that the ABI divides up and uses for various purposes. The ARM and RISC-V CPUs have 32 registers and that is plenty for even the most complex programs. The programs load data into these registers and manipulate that data 64-bits at a time. Memory is used for the runtime stack, program code and data and the program heap.

As stated earlier, the 6502 has 3 8-bit registers:

* A: the accumulator. This is used for all arithmetic operations
* X and Y: index registers, used for indexing memory from a base address

All 6502 programs use a special page[[4]](#footnote-4) of memory called zero-page. Instructions can use the memory in zero-page almost like additional registers. You can perform most operations directly on data held in zero-page locations. It’s slower than operating on data in the accumulator register, but you have 256 bytes available, effectively 256 8-bit registers to play with.

When a procedure call is made, the 6502 pushes the return address following the JSR instruction onto the 6502 stack, which is a page of memory at address 0x100…0x1ff (page 1). This isn’t very much memory and allows a maximum of about 122[[5]](#footnote-5) nested function calls before it overflows.

# Memory layout

A 6502 can address a maximum of 64K of memory (it has a 16-bit address bus). Typically, the memory will be divided into ROM and RAM, with ROM at the high addresses and RAM at the lower. At boot time, the upper 6 bytes of memory must be ROM as it contains the reset vector and two interrupt vectors. The computer is free to remap the memory using hardware to provide banked memory.

This ABI uses a contiguous memory layout for a program using the ABI. This does not preclude any bank switching schemes.

The two bottom pages of memory are used zero-page and the 6502 stack. Following those, a 6502 computer usually has a few pages of scratch space for buffers and other OS data. The program starts just above the OS scratch space and extends up to the end of its static data.

The top of memory contains the OS ROM, which is usually 16K or less but it’s up to the OS and hardware where it actually starts. It’s also possible that the OS ROM is paged out and replaced by RAM if the hardware allows it.

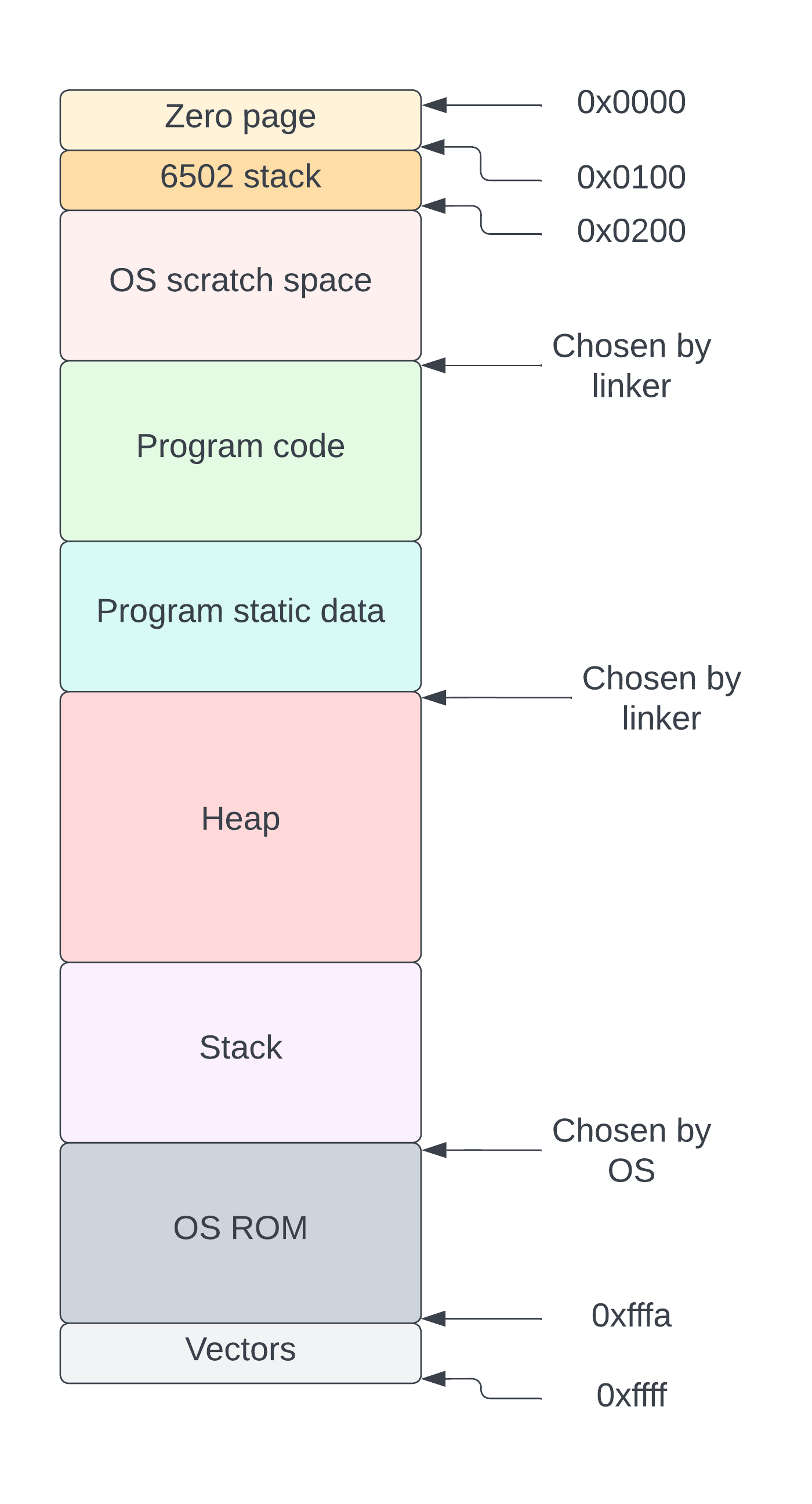
Between the top of the program memory and the bottom of the OS ROM is the program heap and stack. The stack grows down and the heap grows up. This ABI does not specify how the heap space is managed but it will typically be done using a *malloc/free* system.

The linker is free to choose the address for the start of the program memory and the end of the program data. The end of the program data shall be available to the program in the linker-provided global symbol **\_end** and its value shall be the first byte above the program data.

The start address for the program shall be in the global symbol **\_start**, which is usually provided by the runtime support library, but can be provided by the linker.

As an example, the following values might be appropriate for a C program:

* Program code start address: 0x0800
* Highest address for stack: 0xbfff
* OS ROM start address: 0xc000



# Zero-page Registers

We will use some zero-page memory for pseudo-registers for our ABI. Most modern CPUs have fixed size CPU registers (32 or 64-bits wide) and all instructions use these registers. If we were to define a set of 64-bit wide “registers” and use them for all operations we would waste a lot of memory and CPU. Instead, let’s define 5 sets of “registers” in zero page, each of differing sizes, one set for each of the sizes we encounter in a C program:

* 8 8-bit registers (called b0…b7)
* 16 16-bit registers (called i0…i15)
* 8 32-bit registers (called l0...l7)
* 4 64-bit registers (called x0…x3)
* 5 32-bit floating point registers (f0…f3)

In addition, we also need the following housekeeping registers:

* A 16-bit frame pointer (fp)
* A 16-bit stack pointer (sp)
* A 16-bit register to point to the location for a function result
* 60 bytes of runtime library scratch space

All multi-byte registers are in little endian format. The zero-page memory layout is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Start address** | **End Address** | **Name** | **Purpose** |
| 0x00 | 0x07 | b0…b7 | Single byte registers |
| 0x08 | 0x27 | i0…i15 | 2-byte registers |
| 0x28 | 0x47 | l0…l7 | 4-byte registers |
| 0x48 | 0x67 | x0…x3 | 8-byte registers |
| 0x68 | 0x77 | f0…f3 | Single precision floating point registers |
| 0x78 | 0x79 | sp | Stack pointer |
| 0x7a | 0x7b | fp | Frame pointer |
| 0x7c | 0x7d | result | Address to store function result |
| 0x7e | 0xba |  | Scratch space (60 bytes) |
| 0xbb | 0xff |  | Available |

The space from 0xbb to 0xff (68 bytes) is not used by the ABI and is available for operating system use.

### Callee and caller saved registers

Akin to machine registers in a modern CPU, the zero-page registers are a shared resource. We could specify that all registers are saved by the called procedure and restored on return, but it is customary to allow for some scratch registers that do not need to be saved. Compilers can work with these rules to ensure that the value of registers that need to be preserved over a call is preserved.

If a register is specified as “*callee saved*”, it means that if a procedure uses that register, it must ensure that the value of the register is saved on procedure entry and restored on procedure exit. A “*caller saved*” register is also known as a scratch register and its value does not need to be preserved over a call. If the caller needs to keep its value of a call, it needs to save it.

|  |  |
| --- | --- |
| **Registers** | **Saving strategy** |
| b0…b1 | Caller |
| b2…b7 | Callee |
| i0…i3 | Caller |
| i4…i15 | Callee |
| l0…l1 | Caller |
| l2…l7 | Callee |
| x0 | Caller |
| x1…x3 | Callee |
| f0 | Caller |
| f1…f3 | Callee |
| sp | Callee |
| fp | Callee |
| result | Callee |

## C type mappings

The 6502 isn’t a big CPU and we need to take this into account when deciding on the sizes used for C types. On a modern CPU, an **int** is more or less guaranteed to be 32-bits wide, but if we use 4 bytes for every **int** in a C program, we are wasting a lot of memory and CPU (each add operation would take 25 bytes of instructions). It’s more reasonable to use 16-bits for an **int**.

The following table shows the sizes for each of the C types:

|  |  |  |
| --- | --- | --- |
| **C Type** | **Size in bits** | **Register type** |
| char | 8 | b |
| short | 16 | i |
| int | 16 | i |
| long | 32 | l |
| long long | 64 | x |
| float | 32 | f |
| double | 32 | f |
| pointers | 16 | i |

Pointers are naturally 16 bits long since the address space is only 64K. Both **float** and **double** are single precision IEEE754 floating point numbers. It has been very common to not support 64-bit double precision floating point on a 6502 since it’s not worth the extra CPU time and memory[[6]](#footnote-6).

Although 16-bit integers are suitable for most programs, 32 and 64-bit integers are necessary for some programs. It would be onerous for the programmer to be limited to 16 bits for integers.

# Runtime stack

The runtime stack starts at an address chosen to be the top of RAM and grows downwards in memory. It is a ‘full descending” stack meaning that the stack pointer points to the last occupied space in the stack. The stack pointer is in zero-page address 0x78 and 0x79 (2 bytes, little endian) and is initialized to the first address of the OS ROM at program start.

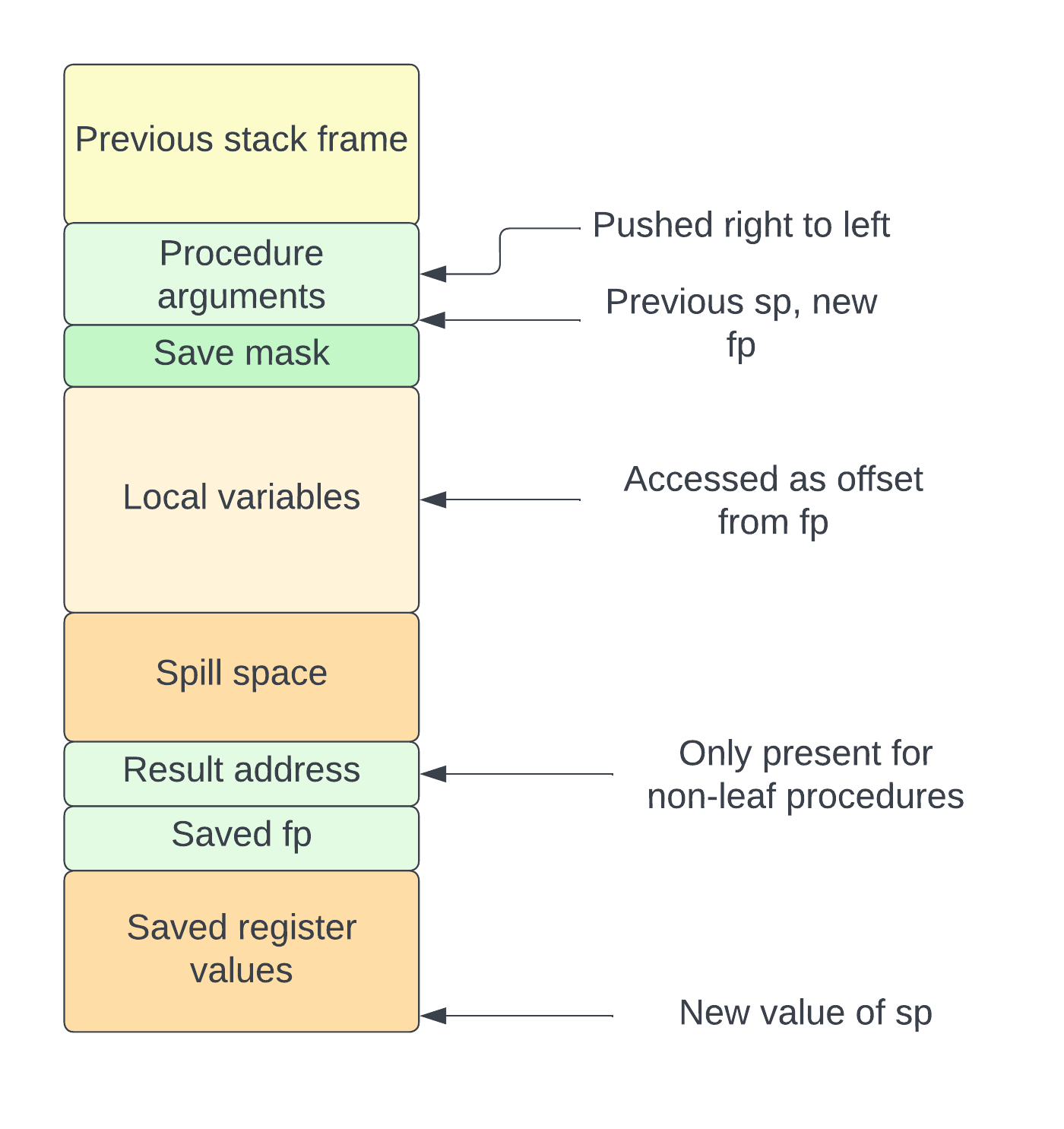
The stack pointer is decremented for each procedure call allowing space on the stack for that procedure’s stack frame and local variables. The frame pointer register (fp) is in zero page at address 0x7a and 0x7b and contains the address immediately above the current procedure’s stack frame. This is the address of the first procedure argument (if any).

There are no artificial limits to the size of the stack or any stack frame. There are practical limits though as you don’t want your stack to grow into the program’s static data or code. Remember that the 6502 doesn’t have any way to write-protect memory.

## The Stack Frame

Upon procedure entry, a stack frame is set up for that procedure. This includes:

* The caller’s frame pointer
* Register save mask
* Address for result
* Local variables
* Spill space
* Callee saved register values.

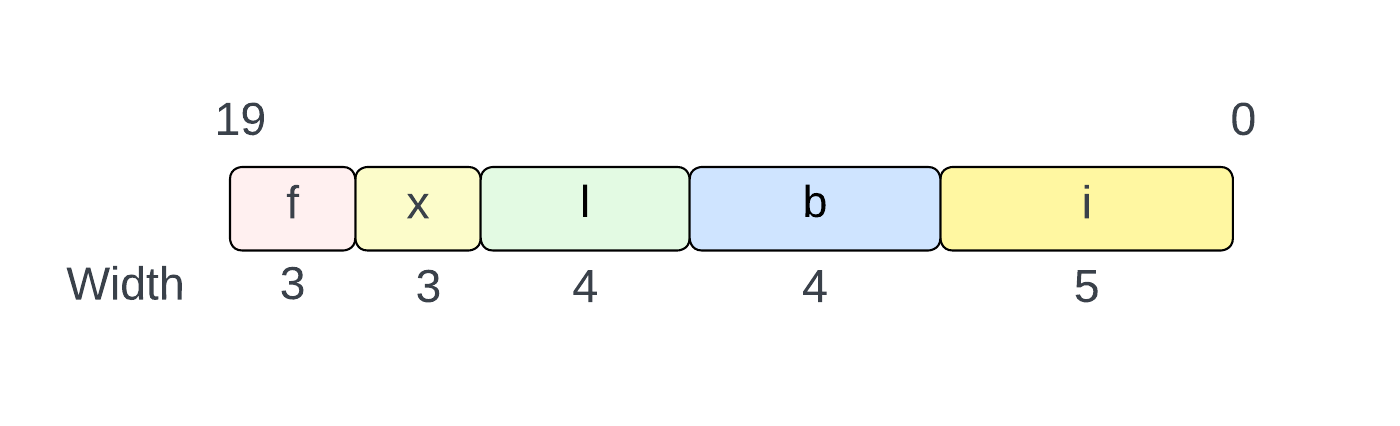


The arguments for the procedure are pushed onto the stack by the caller and are pushed right to left, with the leftmost argument being at the address held in **sp** when the procedure is called. The save mask is discussed in the next section. Local variables are on the stack and are accessed from **fp** with a negative offset. The spill space is used to save the value of registers when an expression is too complex and needs more registers than are available. The **result** address holds the address where the caller wants the result stored. The value of **fp** when the procedure is called is saved at the top of the stack and the new value of **fp** for the called procedure is the value of **sp** when called.

## The Save Mask

The save mask is 24 bits long and contains information about what registers need to be saved by the procedure and restored before it returns. The registers are saved to the local stack frame just below the spill area.

The save mask consists of 5 fields, each of which corresponds to a register set and contains the number of registers of that type to save. Only callee saved registers are saved. It is ordered in little-endian format and is laid out like this:



The register values are saved in variable sized regions on the stack. The layout of the saved registers is implementation defined.

# Procedure entry

When a procedure is called, all the arguments (if any) are pushed onto the stack by the caller and **sp** points to the leftmost argument. If the procedure returns a value, the machine registers X and Y are set to the address into which the argument will be stored. This will normally be a zero-page location but could be any address. The Y register holds the upper 8 bits of the address and X holds the lower 8 bits. If the result is to be placed in a zero-page location (a register), Y will have the value 0.

The stack frame size for the procedure is the amount of memory allocated for all known data excluding the 3 bytes of save mask, which may or may not be present in the stack frame. It also does not include the size of the called saved registers. The actual amount of data allocated from the stack is:

frame\_size + 3 + <space for saved registers>

The entry sequence for regular procedure whose stack frame size is less than 257 bytes and returns a value is as follows:

STX result+0

STY result+1

LDX #frame\_size

JSR \_\_enter

.byte <m0>,<m1>,<m2>

The address into which the result will be stored is saved into zero page and the subroutine **\_\_enter** is called. Before the call to **\_\_enter**, the X register is loaded with the stack frame size. The next 3 bytes after the call to **\_\_enter** contain the save mask in little endian order.

For a stack frame greater than 256 bytes, the sequence is modified to be:

STX result+0

STY result+1

LDX #frame\_size & 0xff

LDY #(frame\_size >> 8) & 0xff

JSR \_\_enter+2

.byte <m0>,<m1>,<m2>

Notice that the call is to **\_\_enter+2**.

The **\_\_enter** subroutine is responsible for creating the stack frame and setting the new values of **fp** and **sp**. The **\_\_enter** subroutine may be implemented as follows[[7]](#footnote-7).

.set \_\_t0 0x7e

.set \_\_t1 0x7f

.set \_\_t2 0x80

.set \_\_t3 0x81

.global \_\_enter

\_\_enter:

LDY #0

// Entry point for >256 bytes on stack frame

STX \_\_t0

STY \_\_t1

// Save old sp

LDA \_\_sp

PHA

LDA \_\_sp+1

PHA

// Store save mask in frame.

// First make room on stack by decrementing \_\_sp by 3

JSR save\_mask\_space

// 2 extra bytes have been pushed onto the stack.

TSX

LDA 0x103,X // Load LO byte

STA \_\_t2 // Copy to temp addr

LDA 0x104,X // Load HI byte

STA \_\_t3

INC \_\_t2 // JSR puts return address -1 on stack.

BNE enter\_skip

INC \_\_t3

enter\_skip:

// Store save mask (24 bits)

LDA (\_\_t2)

STA (\_\_sp)

LDY #1

LDA (\_\_t2),Y

STA (\_\_sp),Y

INY

LDA (\_\_t2),Y

STA (\_\_sp),Y

// Decrement sp by frame size

SEC

LDA \_\_sp

SBC \_\_t0

STA \_\_sp

LDA \_\_sp+1

SBC \_\_t1

STA \_\_sp+1

// Store \_\_result and \_\_result+1 in frame.

LDY #3

LDA \_\_result+1

STA (\_\_sp),Y

DEY

LDA \_\_result

STA (\_\_sp),Y

DEY

// Store old fp.

LDA \_\_fp+1

STA (\_\_sp),Y

LDA \_\_fp

STA (\_\_sp)

// Make new fp (old sp, prior to decrement)

PLA

STA \_\_fp+1

PLA

STA \_\_fp

// Return to address after save mask (\_\_t2 + 2)

// The save mask is 3 bytes but we set the return address to

// one byte less than the next instruction as the RTS will add one

// to it before jumping to it.

enter\_save\_regs:

TSX

CLC

LDA \_\_t2

ADC #2

STA 0x101,X

LDA \_\_t3

ADC #0

STA 0x102,X

// Save registers on the stack (\_\_t2,\_\_t3 contains address of save mask).

JMP \_\_save\_regs

// Makes room for save mask by decrementing sp.

save\_mask\_space:

SEC

LDA \_\_sp

SBC #3

STA \_\_sp

LDA \_\_sp+1

SBC #0

STA \_\_sp+1

RTS

The subroutine **\_\_save\_regs** is responsible for saving the callee save registers onto the stack. It is passed the address of the save mask in 0x80,0x81. It saves the required registers by decrementing **sp** to make space and copying the register’s value onto the stack. After the subroutine is done, **sp** will have been updated to point to the bottom of the saved register values.

If the procedure does not return a value (a *void* function in C, for example), the values of X and Y to specify the result address are not needed and can be omitted.

# Procedure exit

Exiting a procedure involves restoring the values of any called saved registers and removing the stack frame from the stack. The value of **fp** is restored to the value it had on entry to the procedure. Like the entry sequence, the frame size for the procedure does not include the 3 bytes of save mask or the space for the callee saved registers The exit code for a procedure that uses less than 254 bytes of stack space is as follows:

LDY #frame\_size+3

JSR \_\_leave

Notice that the frame size is stored in the X and Y registers in the opposite order from the entry sequence, with Y holding the low byte and X holding the upper byte. This makes the **\_\_leave** subroutine a little faster. Also, notice the Y register is loaded with 3 greater than the frame size, to account for the save mask[[8]](#footnote-8).

For a procedure that uses more than 253 (253+3 = 256) bytes of stack frame, the exit sequence is modified like the entry sequence was:

LDY #(frame\_size+3) & 0xff

LDX #((frame\_size+3) >> 8) & 0xff

JSR \_\_leave+2

The **\_\_leave** subroutine might be coded as:

\_\_leave:

LDX #0

// Entry point for >256 bytes on stack frame

PHY

JSR \_\_restore\_regs

// Load old fp.

LDA (\_\_sp)

STA \_\_fp

LDY #1

LDA (\_\_sp),Y // Y = 1

STA \_\_fp+1

INY

// Load result

LDA (\_\_sp),Y // Y = 2

STA \_\_result

INY

LDA (\_\_sp), Y // Y = 3

STA \_\_result+1

// Increment sp

CLC

PLA

ADC \_\_sp

STA \_\_sp

TXA

ADC \_\_sp+1

STA \_\_sp+1

RTS

The subroutine **\_\_restore\_regs** is responsible for reloading the values of the callee saved registers back from their saved locations on the stack. The save mask is on the stack at the address **fp-3**. It also preserves the value of the X register but not Y or A.

It is important not to restore the value of a callee saved register if that register is used as the result. How this is achieved is implementation defined.

# Entry and exit optimizations

Some procedures do not call other procedures and are called **leaf** procedures. For these, the entry sequence and stack frame are allowed to omit the saved result address since it cannot be overwritten inside the procedure.

If the procedure doesn’t use any callee saved registers, it is permissible to omit the save mask completely.

# Calling a procedure

Calling a procedure is done using a JSR instruction. This saves the return address on the 6502 stack (in page 1) so we get around 120 nested calls before the stack overflows.

If the procedure returns a value, we must pass the address into which the value is to be stored in the X and Y registers with the low byte in X and the high byte in Y. Usually, the address will be stored in a zero-page register, but this isn’t required.

The following 6502 code calls a procedure:

LDX #value & 0xff

LDY #(value >> 8) & 0xff

JSR procedure

For example, to call the function **printf** and put the result in the register **i4**:

LDX #\_\_i4

LDY #0

JSR printf

## Arguments and stack management

All procedure arguments are pushed onto the runtime stack before the call is made and the stack pointer is incremented back up again after the call is done. As is normal for a C language ABI, all integers less than the size of an **int** are promoted to **int** before being pushed. Any value greater than the size of an **int** is pushed intact. There are no alignment requirements. The arguments are pushed from right to left, with the leftmost argument at the address held in **sp**.

Pushing a value onto the stack requires that the stack pointer (**sp**) be decremented by the size of the value being pushed and the value copied into the address referenced by **sp**. This could be done using inline code, but since it’s a common operation, it is suggested that runtime subroutines be provided to push various sizes of values onto the stack. For example, a set of runtime subroutines could be provided, one to push each register onto the stack[[9]](#footnote-9):

JSR \_\_push\_i0

JSR \_\_push\_f2

When the procedure returns, all the pushed arguments must be removed from the stack. This is done simply by incrementing **sp** by size of all the arguments that were pushed. While inline code such as the following is possible:

CLC

LDA \_\_sp

ADC #stack\_size & 0xff

STA \_\_sp

LDA \_\_sp+1

ADC #(stack\_size >> 8) & 0xff

STA \_\_sp+1

This is a common operation, and the above code occupies 13 bytes, so it is suggested that a set of subroutines be provided to increment the stack pointer. For example, you could provide a runtime subroutine called **\_\_incsp2** that increments the stack pointer by 2, and another called **\_\_incspX** that takes the amount to increment by in X.

# Procedure runtime

While a procedure is executing it uses registers in zero-page, accesses variables both from the stack and from static memory, makes procedure calls and executes intrinsic runtime subroutines.

## Loading variables and arguments

Arguments to the procedure are stored on the stack at addresses above the frame pointer. Local automatic variables are on the stack below the frame pointer. Static variables are at fixed locations in memory.

To access an argument, the program adds an offset to **fp**, giving the address of the argument. It can then copy from that address into a zero-page register. The best way to do this uses a helper subroutine to perform the addition and copy into a register.

For example, if we have a 16-bit argument at offset 13 from the frame pointer and want to load that into register **i5**, we can load the offset into the X register and call a helper subroutine to perform the addition and load, as follows:

LDX #13

JSR \_\_arg\_value2\_i5

The helper subroutine **\_\_arg\_value2\_i5** means read the 2-byte argument at offset X from **fp** and store it in **i5**.

Alternatively, we could do it with inline code as follows:

CLC

LDA \_\_fp

ADC #13

STA \_\_t0

LDA \_\_fp+1

ADC #0

STA \_\_t0+1

LDA (\_\_t0)

STA \_\_i5

LDY #1

LDA (\_\_t0),Y

STA \_\_i5+1

This code is 23 bytes long and the example with the helper subroutine is 5 bytes long. The helper version will execute slightly slower as a JSR instruction is 7 cycles just to get to executing the code.

If there are more than 256 bytes of arguments, it would be appropriate to load both X and Y with the offset and call a different helper subroutine that handles a 16-bit offset.

Likewise, to load a local variable into a register, we subtract an offset from **fp** and copy from that address into a zero-page register. For example, to load a 32-bit variable into register **l3** (ell-3) from offset 121:

LDX #121

JSR \_\_var\_value4\_l3

It is also common to calculate the address of an argument or local variable and put that in a register. To do that, add or subtract an offset from **fp**. That can be done inline, or with a helper function. For example:

LDX #4

JSR \_\_arg\_addr\_i2

This would calculate the address of the argument at offset 4 from **fp** and put that address in **i2**.

The inline code for this is reasonably easy and small:

SEC

LDA \_\_fp

SBC #4

STA \_\_i2

LDA \_\_fp+1

SBC #0

STA \_\_i2+1

It’s still bigger than the helper version (13 bytes as opposed to 5).

Loading a static variable is just a load from a known address. The address is fixed when the linker produces the executable output.

Variables can also be stored in zero-page registers if the compiler wants to. This is much more efficient but it is more difficult to generate the code.

## Returning a value

A procedure that returns a value is told where to put that value by the caller. This is different from other ABIs that always return values in a register. On procedure entry, X and Y registers contain the address into which the value will be placed (X= low byte, Y = high byte). The size and type of the value are defined by the return type of the procedure.

The zero-page location **result** (and **result**+1) is available for storing the result address. It is also stored in the stack frame.

For example, a leaf procedure defined as:

int Foo(void) {

return 0;

}

Could be coded as follows (the name **\_\_result** is to prevent clashes with external assembler variable)

Foo:

STX \_\_result+0

STY \_\_result+1

LDA #0

STA (\_\_result)

LDY #1

STA (\_\_result),Y

RTS

## Variable arguments

A variadic procedure can take any number of arguments. The standard header <stdarg.h> defines some macros that enable access to the arguments passed to the procedure.

The type **va\_list** holds the address of the next argument that can be read using **va\_arg**. For the 6502, the **va\_list** type is defined as:

typedef void\* va\_list;

The **va\_start** macro initializes the **va\_list** to the address above its argument. The **va\_start** macro is defined as:

#define va\_start(ap, param)

If param is an *int* at offset 0 above **fp**, and **ap** is in **i7,** this could be coded as:

CLC

LDA \_\_fp

ADC #2

STA \_\_i7

LDA \_\_fp+1

ADC #0

STA \_\_i7+1

The address of the first argument above **param** is at offset 2 above **fp**.

To retrieve the value of an argument, we use the address in the **va\_list** as the base address of the argument, then we copy the value of the argument into the destination and increment the **va\_list** to the next argument. Each argument whose size is less than an *int* (2 bytes) is stored as an *int*. Anything larger is stored as its native size.

For example, if we want an *int* argument to be stored in register **i12**, with the **va\_list** in **i3**, this could be coded as:

LDA (\_\_i3) // Load first byte of argument

STA \_\_i12

LDY #1

LDA (\_\_i3),Y // Second byte of argument

STA \_\_i12+1

CLC

LDA \_\_i3

ADC #2 // Inc va\_list by 2

STA \_\_i3

LDA \_\_i3+1

ADC #0

STA \_\_i3+1

## Long jumps

The standard header <setjmp.h> defines facilities to jump from a called procedure to one of its callers. It does this by saving the state of the registers in a buffer at the call site, and then restoring them at the jump site.

The type **jmp\_buf** holds the caller site state and is defined in this ABI as:

struct \_\_jmp\_buf {

char b[6];

int i[12];

long l[6];

long long x[3];

float f[3];

int sp;

int fp;

int result;

char machine\_sp;

int retaddr;

};

typedef struct jmp\_buf {

struct \_\_jmp\_buf buf;

} jmp\_buf[1];

The reason for the encapsulated **buf** member is due to the C standard requiring **jmp\_buf** to be passed by reference and not by value.

The members of **\_\_jmp\_buf** are:

|  |  |
| --- | --- |
| **Member** | **Purpose** |
| b | Single byte callee saved registers |
| i | 2-byte callee saved registers |
| l | 4-byte callee saved registers |
| x | 8-byte callee saved registers |
| f | IEEE754 single precision callee saved registers |
| sp | Stack pointer register |
| fp | Frame pointer register |
| result | Result address register |
| machine\_sp | 6502 S register (stack pointer) |
| retaddr | Return address from 6502 stack |

The **setjmp** and **longjmp** functions are used in the following way:

static jmp\_buf state;

void Foo(void) {

longjmp(state, 1234);

}

void Bar(void) {

Foo();

}

void Baz(void) {

int x;

if ((x = setjmp(state)) == 0) {

// Return from setjmp.

Bar();

} else {

printf("value is %d\n", x);

}

}

When the function *Baz* is called, it calls **setjmp** to save the current state in the static variable **state**. If this returns the value 0, we know the return was from the **setjmp** call itself, so we call the function *Bar*, which calls *Foo*. *Foo* executes a **longjmp** on the state buffer with the value 1234, which is returned from **setjmp**.

### Saving the state (setjmp)

To save the state at the call site, the **jmp\_buf** is populated with the current values in zero-page and from the 6502 stack. This is done by the **setjmp** function.

Care must be taken when writing **setjmp** because the state being saved must be from the current function and not the **setjmp** function itself. The **longjmp** function will cause a return from **setjmp**. The **setjmp** function returns the value 0 when the return is from the call to **setjmp** and something other than zero when the return is from **longjmp**.

Here is an example that could be used for the 6502 code for **setjmp**. It’s reasonably complex and overwrites the caller saved register **b0**. It also uses some scratch space.

reg\_offsets:

.byte \_\_b0+2, \_\_i0+8, \_\_l0+8, \_\_x0+8, \_\_f0+4

reg\_sizes:

.byte 6\*1, 12\*2, 6\*4, 3\*8, 3\*4

// Entry:

// sp,sp+1: address of jmp\_buf

// X,Y: address for result.

setjmp:

STX \_\_result

STY \_\_result+1

// Copy all registers into the jmp\_buf.

LDA #0

STA \_\_b0 // Register set index.

// Get jmp\_buf from stack and store in \_\_t0,\_\_t1,

LDA (\_\_sp)

STA \_\_t0

LDY #1

LDA (\_\_sp),Y

STA \_\_t1

setjmploop:

LDX \_\_b0

LDA reg\_offsets,X

LDY reg\_sizes,X

PHY

TAX

JSR saveregs

// Increment \_\_t0,\_\_t1 by size of registers stored.

PLA

CLC

ADC \_\_t0

STA \_\_t0

LDA \_\_t1

ADC #0

STA \_\_t1

// Next register set.

INC \_\_b0

LDA \_\_b0

CMP #5

BNE setjmploop

// t0,t1 points to address after registers.

// Save sp, fp and result.

LDY #0

LDA \_\_sp

STA (\_\_t0),Y

INY

LDA \_\_sp+1

STA (\_\_t0),Y

INY

LDA \_\_fp

STA (\_\_t0),Y

INY

LDA \_\_fp+1

STA (\_\_t0),Y

INY

LDA \_\_result

STA (\_\_t0),Y

INY

LDA \_\_result+1

STA (\_\_t0),Y

INY

// Save machine sp

TSX

TXA

STA (\_\_t0), Y

INY

// Save return address

LDA 0x101,X

STA (\_\_t0), Y

INY

LDA 0x102,X

STA (\_\_t0), Y

// Return 0.

LDA #0

STA (\_\_result)

LDY #1

STA (\_\_result),Y

RTS

// Save a set of registers:

// Entry:

// X: start index of registers

// Y: number of bytes to copy

// t0,t1: address to copy to

saveregs:

STY \_\_t2

LDY #0

saveloop:

LDA 0,X

STA (\_\_t0),Y

INX

INY

CPY \_\_t2

BNE saveloop

RTS

### Making the jump (longjmp)

The longjmp function restores the state of the system from the saved state buffer and causes the **setjmp** call to return the value it is passed as the second argument. If the return value is zero, it is incremented to 1 to prevent confusion about the result of **setjmp**.

Here is a possible implementation of the **longjmp** function in 6502 assembly language:

// Entry:

// sp,sp+1: address of jmp\_buf

// sp+2,sp+3: value to return from function.

longjmp:

STX \_\_result

STY \_\_result+1

// Copy all registers from jmp\_buf.

LDA #0

STA \_\_b0 // Register set index.

// Get jmp\_buf from stack and store in \_\_t0,\_\_t1,

LDA (\_\_sp)

STA \_\_t0

LDY #1

LDA (\_\_sp),Y

STA \_\_t1

longjmploop:

LDX \_\_b0

LDA reg\_offsets,X

LDY reg\_sizes,X

PHY

TAX

JSR restoreregs

// Increment \_\_t0,\_\_t1 by size of registers stored.

PLA

CLC

ADC \_\_t0

STA \_\_t0

LDA \_\_t1

ADC #0

STA \_\_t1

// Next register set.

INC \_\_b0

LDA \_\_b0

CMP #5

BNE longjmploop

// t0,t1 points to address after registers.

// Restore result.

LDY #4

LDA (\_\_t0),Y

STA \_\_result

INY

LDA (\_\_t0),Y

STA \_\_result+1

// Copy return value to result.

LDY #2

LDA (\_\_sp),Y

STA (\_\_result)

INY

LDA (\_\_sp), Y

LDY #1

STA (\_\_result),Y

// Check for zero and return 1 if so.

DEY

ORA (\_\_result),Y

BNE result\_not0

LDA #1

STA (\_\_result),Y

result\_not0;

// Restore sp, fp.

LDY #0

LDA (\_\_t0),Y

STA \_\_sp

INY

LDA (\_\_t0),Y

STA \_\_sp+1

INY

LDA (\_\_t0),Y

STA \_\_fp

INY

LDA (\_\_t0),Y

STA \_\_fp+1

// Resore machine sp

LDY #6

LDA (\_\_t0), Y

TAX

TXS

INY

// Restore return address

LDA (\_\_t0), Y

STA 0x101,X

INY

LDA (\_\_t0), Y

STA 0x102,X

// Return from setjmp.

RTS

// Restore a set of registers:

// Entry:

// X: start index of registers

// Y: number of bytes to copy

// t0,t1: address to copy from

restoreregs:

STY \_\_t2

LDY #0

restoreloop:

LDA (\_\_t0),Y

STA 0,X

INX

INY

CPY \_\_t2

BNE restoreloop

RTS

# ELF mappings

Although not required, if this ABI is used with the Executable and Linkable Format (ELF) object file format, the following definitions apply:

The machine type used is **EM\_6502** or **EM\_65C02** and its value is **0x1966** (6502). Either ELF64 or ELF32 may be used (neither really applies, so it doesn’t matter).

Dynamic linking is not possible on the 6502 architecture, so only static linking is permitted.

The relocations available to the linker are as follows:

#define R\_W65C02\_JSR 1 // Call direct to symbol.

#define R\_W65C02\_JMP 2 // Move symbol address to reg.

#define R\_W65C02\_DATA16 3 // 16-bit data.

#define R\_W65C02\_DATA32 4 // 32-bit data.

#define R\_W65C02\_DATA64 5 // 64-bit data.

#define R\_W65C02\_BYTE0 6

#define R\_W65C02\_BYTE1 7

#define R\_W65C02\_BYTE2 8

#define R\_W65C02\_BYTE3 9

#define R\_W65C02\_BYTE4 10

#define R\_W65C02\_BYTE5 11

#define R\_W65C02\_BYTE6 12

#define R\_W65C02\_BYTE7 13

#define R\_W65C02\_ADD8 14 // Add 8 bits.

#define R\_W65C02\_ADD16 15 // Add 16 bits.

#define R\_W65C02\_ADD32 16 // Add 32 bits.

#define R\_W65C02\_ADD64 17 // Add 64 bits.

#define R\_W65C02\_SUB8 18 // Subtract 8 bits.

#define R\_W65C02\_SUB16 19 // Subtract 16 bits.

#define R\_W65C02\_SUB32 20 // Subtract 32 bits.

#define R\_W65C02\_SUB64 21 // Subtract 64 bits.

In the following table, **S** is the value of the referenced symbol, **A** is the addend in the relocation, **addr** is the address being relocated.

|  |  |  |
| --- | --- | --- |
| **Name** | **Value** | **Operation** |
| R\_W65C02\_JSR | 1 | JSR <addr>: addr = S + A |
| R\_W65C02\_JSR | 2 | JMP <addr>: addr = S + A |
| R\_W65C02\_DATA16 | 3 | addr[0:15] = S + A |
| R\_W65C02\_DATA32 | 4 | addr[0:31] = S + A |
| R\_W65C02\_DATA64 | 5 | addr[0:63] = S + A |
| R\_W65C02\_BYTE0 | 6 | addr[0:7] = (S+A) & 0xff |
| R\_W65C02\_BYTE1 | 7 | addr[8:15] = (S+A) & 0xff |
| R\_W65C02\_BYTE2 | 8 | addr[16:23] = (S+A) & 0xff |
| R\_W65C02\_BYTE3 | 9 | addr[24:31] = (S+A) & 0xff |
| R\_W65C02\_BYTE4 | 10 | addr[32:39] = (S+A) & 0xff |
| R\_W65C02\_BYTE5 | 11 | addr[40:47] = (S+A) & 0xff |
| R\_W65C02\_BYTE6 | 12 | addr[48:55] = (S+A) & 0xff |
| R\_W65C02\_BYTE7 | 13 | addr[55:63] = (S+A) & 0xff |
| R\_W65C02\_ADD8 | 14 | addr[0:7] += (S+A) & 0xff |
| R\_W65C02\_ADD16 | 15 | addr[0:15] += (S+A) & 0xffff |
| R\_W65C02\_ADD32 | 16 | addr[0:31] += (S+A) |
| R\_W65C02\_ADD63 | 17 | addr[0:63] += (S+A) |
| R\_W65C02\_SUB8 | 18 | addr[0:7] -= (S+A) & 0xff |
| R\_W65C02\_SUB16 | 19 | addr[0:15] -= (S+A) & 0xfffff |
| R\_W65C02\_SUB32 | 20 | addr[0:31] -= (S+A) |
| R\_W65C02\_SUB64 | 21 | addr[0:63] -= (S+A) |

1. To be accurate, this is for Western Design Center’s 65C02, which is really the only one you can buy these days. The examples use 65C02 instructions that are not present on the original MOS 6502. [↑](#footnote-ref-1)
2. OK, I cheated, this is the actual output from my C compiler. [↑](#footnote-ref-2)
3. GAS is the GNU Assembler, a free assembler available on most UNIX systems. [↑](#footnote-ref-3)
4. A page is 256 bytes of memory on the 6502 [↑](#footnote-ref-4)
5. 128 actually, but you also need space for interrupts, which use 3 bytes of this stack space. There are 2 interrupt lines. [↑](#footnote-ref-5)
6. When the 6502 was state of the art, most computers used a non-standard floating point format with a 4 byte mantissa. Single precision IEEE754 uses 3 bytes for the mantissa, so it isn’t as accurate as the older formats. However, full double precision floating point isn’t worth it. [↑](#footnote-ref-6)
7. The zero-page register names are prefixed by a double underscore to prevent clashes with any external variables with the same name. [↑](#footnote-ref-7)
8. This is an optimization to prevent an extra subtraction operation in the \_\_leave subroutine. Might as well perform the calculation at compile time rather than at runtime. [↑](#footnote-ref-8)
9. Runtime and 6502 stack overflow checking is not mandated but can be done if desired. [↑](#footnote-ref-9)