## Section 1 / Passing Parameters To Functions

How parameters are passed to functions can be different from OS to OS. This chapter is written to the standard implemented for Linux. It differs from the **calling convention** used on, for example, the Mac in that parameters are principally passed via the scratch registers.

Up to 8 parameters can be passed directly via registers. Each parameter can be up to the size of an address, long or double (8 bytes). If you need to pass more than 8 parameters or you need to pass parameters which are larger than 8 bytes or are `structs`, you would use a different technique described later.

Remember that even large data structures that are passed by reference are, in fact, passed via their base address (as a pointer).

For the purposes of the present discussion, we assume all parameters are `long int` and are therefore stored in `x` registers.

Up to 8 parameters are passed in the scratch registers (of which there are a matching 8). These are `x0` through `x7`. *Scratch* means the value of the register can be changed at will without any need to backup or restore their values.

**This means that you cannot count on the contents of the scratch registers maintaining their value if your function makes any function calls.**

For example:

```
long func(long p1, long p2)                               // 1
{                                                         // 2
    return p1 + p2;                                       // 3
}                                                         // 4
```

is implemented as:

```
func:   add x0, x0, x1                                    // 1
        ret                                               // 2
```

The value of the first parameter (`p1`) is copied into the first scratch register (`x0`). It's an `x` because the parameter's type is `long int`. It is the `0` register, because that is the first scratch register.

The value of the second parameter (`p2`) is copied into the second scratch register (the `1` register) because it is the second argument, and so on.

`Line 1` of the assembly language provides the label `func` to which a `bl` can be made.

`Line 1` also provides the full body of the function - the third argument to `add` is added to the second and the result is put in the first. Thus it is: `x0 = x0 + x1`.

Just as scratch registers are used for passing (up to 8) parameters, the `0` register is used for function returns. In the case of the current code, the result of the addition is already sitting in `x0` so all we do is `ret` on `Line 2`.

**This is an advanced topic:** If you are the author of both the caller and the callee and both are in assembly language, you can play loosey goosey with how you return values. Specifically, you can return more than one value. **But** if you do so, you give up the possibility of calling these functions from C or C++. Maybe you should forget you read this paragraph?

## `const`

Suppose we had:

```
long func(const long p1, const long p2)                        // 1
{                                                              // 2
    return p1 + p2;                                            // 3
}                                                              // 4
```

how would the assembly language change?

Answer: no change at all!

`const` is an instruction to the compiler ordering it to prohibit changing the values of `p1` and `p2`. We're smart humans and realize that our assembly language makes no attempt to change `p1` and `p2` so no changes are warranted.

## Passing Pointers

A pointer is an address of something. The word *pointer* is scary. The words *address of* are not as scary. They mean **exactly** the same thing.

Here is a function which *also* adds two parameters together but this time using pointers to `long int` rather than the values themselves.

```
void func(long * p1, long * p2)                                // 1
{                                                              // 2
    *p1 = *p1 + *p2;                                           // 3
}                                                              // 4
```

`Line 1` passes the *address of* `p1` and `p2` as parameters. That is, the addresses of `p1` and `p2` are passed in registers `x0` and `x1` rather than their contents. The contents of the underlying longs still reside in memory. That is:

- The address of `p1` arrives in `x0`. The value of `p1` still resides in memory.
- The value of `p1` is found in memory at the address specified by the parameter.

`Line 3` *dereferences* the addresses to fetch their underlying values. The values are added together and the result overwrites the value pointed to by `p1`.

Here it is in assembly language:

```
func:   ldr x2, [x0]                                // 1
        ldr x3, [x1]                                // 2
        add x2, x2, x3                              // 3
        str x2, [x0]                                // 4
        ret                                         // 5
```

The `add` instruction cannot operate on values in memory.

With little exception, all the *action* takes place in registers, not memory. Therefore, the underlying values pointed to by the parameters must be fetched from memory.

`Line 1` provides the label to which a use of `bl` can branch with link register.

Remember that up to the first 8 parameters are passed in the 8 scratch registers. Thus, the address of `p1` and the address of `p2` are stored in `x0` and `x1` respectively. `0` and `1` because these are the first two parameters. The `x` form of the `0` and `1` registers are used because the parameters' type are addresses.

- Addresses (pointers) to any type are 64 bits wide and therefore must use `x` registers.
- `long` and `unsigned long` integers are 64 bits wide and ...
- `double` floats are 64 bits wide

`Line 1` also dereferences the address held in `x0` going out to memory and loading (`ldr`) the value found there into `x2`, another scratch register. It's scratch so it doesn't need backing up and restoring.

`Line 2` does the same for `p2`, putting its value in `x3`.

To say this again but differently, the syntax `[` then an `x` register followed by `]` means use the `x` register as an address in RAM. Go to that address and fetch its value. This is a *dereference*.

Why didn't we reuse `x0` and `x1` as in:

```
ldr   x0, [x0]
ldr   x1, [x1]
```

Doing so would be legal but would end in tears.

Doing so would blow away the address of `p1` (and `p2` too). Destroying the address of `p1` would prevent us from copying the result of the addition back into memory since the address to which we would want to store the result of the addition would be gone. Can't have that!

So, as the smart *human*, we decided to use `x2` and `x3` because, well, they're scratch.

`Line 3` performs the addition.

`Line 4` stored the value in `x2` at the address in memory still sitting in `x0`.

**Passing by Reference**

Suppose we had:

```
long func(long & p1, long & p2)                        // 1
{                                                      // 2
    return p1 + p2;                                    // 3
}                                                      // 4
```

how would the assembly language change?

Answer: just a little:

```
func:   ldr x2, [x0]                                   // 1
        ldr x3, [x1]                                   // 2
        add x2, x2, x3                                 // 3
        mov x0, x2                                     // 4
        ret                                            // 5
```

Passing by reference is also an instruction to the compiler to treat pointers a little differently - the differences don't show up here so there the only change to our pointer passing version is how we return the answer.

But wait…

There is a small optimization we can make here:

```
func:   ldr x0, [x0]                                   // 1
        ldr x1, [x1]                                   // 2
        add x0, x0, x1                                 // 3
        ret                                            // 4
```

This time we're not storing anything back to `p1` or `p2` so we can reuse `x0` and `x1` since the addresses they contained aren't needed again. Smart human!

## What If We Need More Than Eight Parameters?

First, do you **really** need to pass more than 8 parameters? **REALLY?**

If for some reason you do, you can pass the first 8 in registers are described above. Beginning with the ninth parameter, these would be passed on the stack.

**REMEMBER THAT ANY ADJUSTMENT TO THE STACK MUST BE DONE IN MULTIPLES OF 16!**

- If you need just 1 byte of stack, the stack pointer must be changed by 16.
- If you need 17 bytes of stack, the stack pointer must be changed by 32 and so on.

Here is a sample function that requires 9 parameters (for who knows what reason):

```
#include <stdio.h>

void SillyFunction(long p1, long p2, long p3, long p4,
                   long p5, long p6, long p7, long p8,
                   long p9) {
    printf("This example hurts: %ld %ld\n", p8, p9);
}

int main() {
    SillyFunction(1, 2, 3, 4, 5, 6, 7, 8, 9);
}
```

This prints:

```
This example hurts my brain: 8 9
```

In assembly language, this program could be written as:

```
        .text                                           // 1
        .global    main                                 // 2
                                                        // 3
SillyFunction:                                          // 4
        str        x30, [sp, -16]!                      // 5
        ldr        x0, =fmt                             // 6
        mov        x1, x7                               // 7
        ldr        x2, [sp, 16]                         // 8
        bl         printf                               // 9
        ldr        x30, [sp], 32                        // 10
        ret                                             // 11
                                                        // 12
main:                                                   // 13
        str        x30, [sp, -16]!                      // 14
        mov        x0, 9                                // 15
        str        x0, [sp, -16]!                       // 16
        mov        x0, 1                                // 17
        mov        x1, 2                                // 18
        mov        x2, 3                                // 19
        mov        x3, 4                                // 20
        mov        x4, 5                                // 21
        mov        x5, 6                                // 22
        mov        x6, 7                                // 23
        mov        x7, 8                                // 24
        bl         SillyFunction                        // 25
        ldr        x30, [sp], 32                        // 26
        ret                                             // 27
                                                        // 28
        .data                                           // 29
fmt:    .asciz    "This example hurts: %ld %ld\n"       // 30
```

Notice how `main()` puts the first 8 parameters into the scratch registers `x0` through `x7` using `Lines 17` to `24`. But first, it put the ninth parameter onto the stack. It did the stack parameter first so that the stack pointer could be manipulated in a scratch register.

After executing `Line 14`, the stack will have:

```
sp + 0    return address for main
sp + 8    zero
```

After executing `Line 16`, the stack will have:

```
sp + 0    9
sp + 8    garbage
sp + 16   return address for main
sp + 24   zero
```

After executing `Line 5`, the stack will have:

```
sp + 0    return address for SillyFunction
sp + 8    garbage
sp + 16   9
sp + 24   garbage
sp + 32   return address for main
sp + 40   zero
```

This means that `Line 8` fetches `p9` from memory and puts its value into x2 (where it becomes the third argument to `printf()`).

## A bit of history

The early Unix kernels would abuse the calling convention to miraculously pass return values back to calling functions. Early versions of C made extensive use of a now obsolete keyword `register`. It was an instruction to the compiler to store a certain variable in a register and not in memory in the code the compiler produced.

Particularly abusive functions would call other functions without passing any actual variables but the parameters would indeed be passed! The coders assumed the compiler would store specific variables in specific registers, avoiding the overhead of using the actual calling convention they themselves defined. Code that did this had to be rewritten once Unix began to be ported to machines beyond the original DEC hardware.

This had the author scratching his head until he figured it out, way way back in the day.

Those were the days when the entire Unix kernel would be printed out to form a stack of paper less than an inch high. The author knows this because Jishnu Mukerji presented such a stack to the author the third time the author asked Jishnu a question about the kernel. He gave the author answers to two questions. On the third question, he handed the author the print out and said: *"All your answers are in here."* The author deeply appreciates Jishnu Mukerji's formative impact on a young undergraduate.