

Subroutines

Announcements

- Lab #1 (Toys) paper copy due today.
- Quicksort officially out today—handout given out last Wednesday.
- **The midterm is one week from today during class.** In order to allow for alternate seating, **we are in a different room: 200-34.**
- The practice midterm goes out today—it's very important to look at it.
- Midterm review session will be this Friday in b08 (12:50 - 1:40).
- Fixups from last time:
 - DC.W @casex-y → @casex+y (mentioned in class)
 - MOVE.W CaseTable(PC, D0) → MOVE.W CaseTable(D0)

Last time

- Control structures
 - if
 - while / for
 - **switch**
- Stacks (mostly finished)

This time

- Finish up stacks
- Subroutines
 - Calling and returning
 - Passing parameters
 - Returning results
 - Local variables
 - Register conventions

Subroutines

The idea of subroutines is another of those concepts that you really can't write interesting programs without. The most important property of subroutines is that you can jump to them and then *jump back* to where you started. The ability to jump back to the location of the call allows you to write a function once and then call it from numerous different locations. Without being modified to know about the calling code, the subroutine can still return control to the correct place when it's done.

Up until now, you've used the `JSR` and `RTS` commands to make calls to and return from subroutines. You used registers (or global variables) to pass and return data and also for local storage. Today, we will go through lots of different ways to accomplish the goals of calling/returning, passing parameters, returning results, and storing local data. We will also learn about different function calling conventions and see what support the 68K gives us for making function calls.

Calling and returning

As we've already said, the most important thing about a subroutine is that you've got to be able to call it and then to return to where you started from. Let's forget about the fact that we've already got `JSR` and `RTS` to do this for us and think of the different ways we could do it ourselves.

Use a register for the return address (method A)

Maybe one of the most intuitive ways to call and return is to store the return address in a register, say `A0`. Then, we can jump to the routine using PC-relative addressing and return from the routine using register-indirect addressing. Here's a code snippet that uses this method:

```

caller:
                LEA    @next, A0
                JMP    Foo
@next:         ...

callee:
Foo:           ...
                ...
                JMP    (A0)

```

Let's look at the pro's and cons for this method:

- pro: simple
- pro: fast (everything is done in registers)
- con: can't do nested calls without more work (callee would have to save and restore `A0`).

Don't scoff—this is a valid way of doing things. However, let's see what else we can think of...

Use global storage space to put your return address (method B)

We can avoid the "con" from above if we allocate some global space for each routine that is used to store the address that the routine should return to:

```

static void *FooRA;

caller:
                LEA    @next, A0
                MOVE.L A0, FooRA
                JMP    Foo
@next:         ...

callee:
Foo:           ...
                ...
                MOVEA.L FooRA, A0
                JMP    (A0)

```

What are the advantages and disadvantages of this method?

- pro: allows nested function calls.
- con: still doesn't allow *recursive* function calls.
- con: slow when compared to register approach.
- con: space grows with number of functions that exist—ideally, you'd like it to grow with the level of nesting.

This is a valid convention, but there's really no reason to use it because there's a better way...

Use a stack for your return address (method C)

Last time, I said that you needed to learn about stacks before you could learn about subroutines. The reason is that stacks provide a great way to make calls. Let's say that A7 is our stack pointer. That means that when we start out, A7 points just below the memory allocated for the stack. We can make function calls like:

```
caller:                                callee:
                                         Foo:  ...
                                         ...
                                         MOVEA.L (A7)+, A0
@next:  ...                             JMP    (A0)
```

A stack is the perfect data structure for subroutine calls because it allows subroutines to be nested and even allows recursion. Let's look at the pro's and cons:

- pro: allows nested *and recursive* function calls
- pro: space grows with the level of nesting
- con: slow when compared to storing the return address in registers

On the 68000, using the stack for return addresses is supported by the instructions JSR and RTS. These instructions do just what we did by hand (well, they don't trash A0...but they do the rest). We can re-write the above code:

```
caller:                                callee:
                                         Foo:  ...
                                         ...
                                         RTS
```

Note that when you use JSR and RTS, you don't specify that you want to use A7 for your stack. A7 is assumed to be the stack. Thus, A7 is *special*. In fact, you can actually refer to it by another name: SP (for stack pointer). Note that although the 68K is designed to use A7 as the stack pointer, space isn't allocated for the stack by the processor itself. It is the application's (or the operating system's) job to allocate memory for the stack and set A7 to point to it.

Which method should you use?

I would advocate that there is really no reason to ever use method B. Methods A and C, however, both have their advantages and disadvantages (you can get around con in method A by storing the return address register on a stack whenever you have nesting). Probably when you're working on the 68K, method C is the way to go. That's because it's supported by the 68K instruction set (JSR/RTS) and is what other people are expecting you to do. However, other instruction sets (like MIPS) provide support for method A. On those processors, you should probably use method A.

Since we're working on the 68K in this class, we will always use JSR/RTS for calling and returning.

Passing parameters

Another important feature of subroutines is that you can pass them values to specify how they should work. Again, there are a number of possible ways to pass parameters. As you'll notice, the different choices map very closely to the different choices for specifying the return address. That's because you can view specifying the return address as just a special case of passing parameters.

For our examples, we will use the following made up function and call:

Function: void Foo (int x, int y, int *a)

Call: Foo (m, 10, &n)

...where m and n are global variables.

Use a registers for parameters (method A)

So far in your assignments, you have (probably) used registers to pass parameters. This is the way that the CS 110 library routines accept parameters and is really the easiest way to pass things to your own routines. Let's look at our example function call:

caller:		callee:	
	MOVE.W m, D0	Foo:	... // use D0, D1
	MOVE.W #10, D1		... // and A0
	LEA n, A0		
	JSR Foo		RTS

As you no doubt have realized from doing your Toys assignment, this is not perfect. Let's look at the pro's and cons (as usual):

- pro: simple
- pro: fast

- con: the number of registers (and thus the number of parameters) is limited. Also, using precious registers for passing parameters means that you have fewer registers for storage.
- con: nesting routines is difficult.

The key reason why our library routines use registers for passing parameters is that they are very simple to understand. You can use the CS 110 library from day 1 (ok: maybe from day 3).

Use global storage space (parameter areas) to pass parameters (method B)

If you found yourself running out of registers on assignment 1, you may have used global storage space to help alleviate your pain. Here's an example (though we probably won't actually go over this in class):

```
static Word Foo1, Foo2;
static void *Foo3;

caller:                                callee:

        MOVE.W    m, Foo1                Foo:    ...    // use Foo1
        MOVE.W    #10, Foo2              ...    // Foo2, Foo3...
        LEA       n, A0
        MOVE.L    A0, Foo3                RTS
        JSR      Foo
```

Pro's and con's are similar to putting the return address in global space.

- pro: still pretty simple
- pro: nesting much easier
- pro: leaves you more registers to work with yourself
- con: doesn't easily allow *recursive* function calls
- con: slow when compared to registers
- con: space grows with number of functions that exist.

Like putting the return address in global space, there's really not a good reason to use this because you can:

Use a stack for your parameters (method C)

Just as we found a stack useful for putting return addresses on, it is also useful to use a stack to pass parameters. Conceivably, we could allocate a separate stack in memory and use some other register as the parameter stack pointer. However, it turns out that that's not needed. As I mentioned on the previous handout, there is no reason that a stack has to be homogeneous. It can be heterogeneous. In other words, we can mix parameters and return addresses on the same stack as long as we push and pop in the correct order. Let's see an example:

What have we done differently?

- We pushed parameters on in a different order.
- The callee was responsible for cleaning up the stack.

So you're probably asking "who cares?" right about now. Well, it turns out that both of these conventions are actually used. The first one is used by the "C" calling convention to pass parameters. The second one is used by the "Pascal" calling convention. In Metrowerks, the C convention is the one used by default. However, you can specify the Pascal convention by using the `pascal` keyword.

First, let's look at why parameter order matters. It turns out that the Pascal parameter order is pretty much just arbitrary. Whoever decided on the Pascal convention just decided that parameters should be pushed on left to right. What's the difference, right? The C calling convention probably would have used the same convention except that the left-to-right convention doesn't allow for one thing: variable arguments. If you pass parameters left to right, notice that the thing that is on top of the stack is the *last* parameter passed. However, if you pass parameters right to left, the thing on the top of the stack is the *first* parameter passed. The C calling convention was designed to allow you to write functions like `printf`, whose prototype is:

```
int printf (char *format, ...);
```

In order to figure out how many parameters were passed, `printf` must look at the format string. If we were using the Pascal convention, this would be impossible because we wouldn't know how far down the format string was unless we knew how many parameters were passed (which we don't know because we don't yet have the format string). Note that if we wanted to write a function like `printf` using the Pascal convention, we would have to do something like:

```
int printf (... , char *format);
```

Second, let's look at the differences in who cleans up the stack. This time, it turns out that Pascal's convention was actually chosen for a reason. From a code-size perspective, it makes more sense to have the callee clean up the stack. That's because there are always more callers than callees (assuming no functions go unused). That is, each function gets called 1 or more times. Instead of duplicating the cleanup code for every caller, the Pascal convention just has the cleanup code once. If it's so much more efficient, why didn't the C convention also use this method? Again, it has to do with variable argument functions. If the callee is responsible for cleaning up the stack, what happens if the callee doesn't know how much to clean up?

Sidenote: notice that in our example of the Pascal convention, we didn't use RTS. That's because we had to clean off the stack below where the return address was. This is sort of a pain and was fixed in the newer 68K's (the 68010 has an `RTD` instruction).

Stack/register mixture (method E)

As you may be realizing, there are virtually an infinite number of methods for passing parameters. One other interesting method to point out is a mixture of register and stack based passing. You could imagine specifying that D0/D1 and A0/A1 were to be used for the first two data values/address values. Then, any successive values would be pushed on the stack. Something akin to this method is used in the MIPS calling convention. It allows for fast parameter passing if you have a small number of parameters but still is general enough to allow for any number of parameters. Nested calls are accomplished by saving registers.

Which method should you use?

That really depends. If you're really going for speed, you can't beat using registers for passing parameters. However, using C or Pascal calling conventions is certainly the simplest to keep straight in your head (you don't have to worry about register clashes, etc). In any case, if you're trying to interact with code that you didn't write (or you want other code to interact with you), you will be forced to use the conventions defined by the creator of that code. Using standard conventions can also make your code more readable to others.

Returning values

If you remember from my introduction to subroutines, the third thing that we need to figure out how to accomplish with subroutines is returning values. We'll be seeing some common themes again.

This time, we will use the following made up function and call:

Function: int Foo (int x)

Call: y = Foo (10)

...where y is a global variable. I will use stack-based parameter passing...

Use a registers for the return value (method A)

As you probably guessed, the first method that we will look at is using registers to return values. Again, this was the way that the CS 110 library functions worked. Here's the obligatory sample code:

caller:		callee:	
	MOVE.W #10, -(A7)	Foo:	...
	JSR Foo		
	MOVE.W D0, y		// Put result in D0
	ADDQ.L #2, A7		RTS

...and of course, the pro's and con's:

- pro: simple and fast.
- pro: if caller ignores the return value, doesn't have to do anything
- con: only allows one return value (well, you could specify more registers, but...)
- con: return value must be 8-32 bits.
- con: need to move return value out of D0 before next call.

Interestingly enough, this method is used by the C calling convention.

Use global space to put the return value (method B)

Ok, ok...so this is a silly method.

Push the address of a location to store the return value (method C)

In this method, we add an extra parameter to our call (or more than one extra parameter for multiple return values). The extra parameter is just the address to store the result in. It's probably best to push this address on right after the registers.

```
#define rOff          4

    caller:                                callee:
                                         Foo:   ...
                                         MOVE.L  rOff(A7), A0
                                         MOVE.W  result, (A0)
                                         RTS
```

- pro: any number of return values of any size can be specified, as long as the caller and callee agree on their size and number.
- pro: callee puts the value just where you want it.
- con: slow (uses two memory references rather than zero, and one is a long).
- con: forces you to locate x in memory (can't have x completely register based).

This is the method that Metrowerks' C calling convention uses when the return value is larger than 32 bits (like a structure).

Use the stack (method D)

Method C used the stack to store the *address* of the location of the result. However, if we think a little bit, we realize that we can actually use the stack itself to store the return value. We can allocate space on the call stack for the callee to put the result Note: I will do this example using Pascal convention (I've been using C convention for the other ones):

```
#define      rOff      8
```

```
caller:
```

```
    SUBQ.L    #2, A7
    MOVE.W   #10, -(A7)
    JSR      Foo
    MOVE.W   (A7)+, y
```

```
callee:
```

```
Foo:    ...
        MOVE.W  result, rOff(A7)
        MOVE.L  (A7)+, A0
        ADDA.L  #2, A7
        JMP     (A0)
```

- pro: any number of return values of any size can be specified, as long as the caller and callee agree on their size and number.
- con: slow (uses one memory access).

I did this example with Pascal convention for a reason: this method for returning a value is used by the Pascal convention.

Which method should you use?

You're probably sick of this section by now. Methods A, C, and D are all used (A and C are both used by C, D is used by Pascal).

Variables

Now, we need to look at how functions can store local data.

Use registers to store local data

Certainly, you know how to use registers to store data by now. Just decide which registers you're going to use and then use them.

Use global space to store local data

You ought to know how to do this too (if not, you probably didn't follow much of the rest of this lecture). Allocate some global space and store stuff there.

Use the stack to store local data

Ah ha! You thought I was going to say that you should know this by now, didn't you? Well, you're wrong. While it's true that you ought to know the basic ideas behind storing local variables on the stack, there are some tricks that you will certainly want to use if you're really serious about storing stuff on the stack. As usual, we'll look at an example and that will (hopefully) make everything clear.

Our example will look like:

```
Function:  void Foo (int x) {
            int l;
            l = 10;
            ...
        }
Call:      Foo (10)
```

Let's write this using C calling conventions and putting `l` on the stack.

```
#define xOff      6
#define lOff      0
#define varSize   2
```

caller:

```
MOVE.W #10, -(A7)
JSR     Foo
ADDQ.L #2, A7
```

callee:

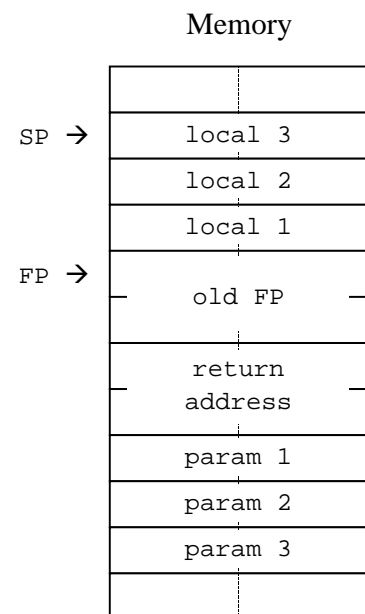
```
Foo:    SUBA.L #varSize, A7
        MOVE.W #10, lOff(A7)
        ...
        ADDA.L #2, A7
        RTS
```

This seems to work fine in the simple case. However, what happens if we want to add another local variable? All of a sudden, the offset of `x` changes. That's because it's now further down the stack. Fine, so we update our `#define`. Now, what happens if we're pushing on parameters for a function call and want to access one of the variables on the stack (or one of our own parameters)? We have to calculate the actual offset by hand (depending on how many parameters we've already pushed on). Not so easy.

The frame pointer

While all of this is possible, it gets to be a pain rather quickly. In order to alleviate this pain, people commonly use what's called a "frame pointer." The frame pointer is set up right at the beginning of a function call. It points to the boundary of the local variables and the parameters. Thus, local variables are negative offsets from the frame pointer and parameters are positive offsets. Typically, `A6` is used for the frame pointer, but this is not required (not like using `A7` for the stack pointer is).

We would like to use the frame pointer in all of our functions. However, this poses a problem. If every function set `A6` to be its own frame pointer when starting, the frame pointer will only be valid until the first function call. After that, it will be completely



wrong. That would make the frame pointer virtually useless. There is a fix, as shown in my diagram. We can store the old frame pointer on the stack when we establish the current frame pointer. Then, when the function ends, we can restore the old frame pointer. This will make it look to the calling function as if nothing changed.

Let's look at our previous example, this time with a frame pointer.:

```

#define      xOff      8
#define      lOff      -2
#define      frameSize 2

caller:

        MOVE.W   #10, -(A7)
        JSR      Foo
        ADDQ.L   #2, A7

        (caller doesn't change!)

callee:

Foo:    MOVE.L   A6, -(A7)
        MOVEA.L  A7, A6
        SUBQ.L   #frameSize, A7
        MOVE.W   #10, lOff(A6)
        ...
        MOVEA.L  A6, A7
        MOVEA.L  (A7)+, A6
        RTS

```

Whoa! Lots of extra overhead. Is it really worth it? It sure is convenient to be able to find all of your local variables and parameters, even if you've added more junk onto the stack. Still, it seems like a lot of extra work. Well...it turns out that the 68000 is here to help you again with the LINK and UNLK instructions, which make everything shorter.

```

#define      xOff      8
#define      lOff      -2
#define      frameSize 2

caller:

        MOVE.W   #10, -(A7)
        JSR      Foo
        ADDQ.L   #2, A7

        (caller doesn't change!)

callee:

Foo:    LINK      #-frameSize, A6
        MOVE.W   #10, lOff(A6)
        ...
        UNLK     A6
        RTS

```

Go CISC! The 68K has an instruction for everything (well...except reversing the bits in a byte).

Stack crawls

An interesting thing about the frame pointer: using it you can always find the return address of the current function, no matter what other junk you've put on the stack. You can also find the old frame pointer. Using the old frame pointer, you can find the return address of the calling function and its old frame pointer. If you have all this information, you can travel back through the stack (following link after link) and figure out all the functions that called you. If you had

some sort of information about which functions were located where, you could use all this information to produce what is known as a stack trace (or stack crawl). It turns out that the PalmDebugger has the ability to do a stack crawl, assuming that you are using the convention of having A6-based frame pointers. Let's look at a stack crawl (the command is `sc`) on my Palm:

```
sc
Calling chain using A6 Links:
A6 Frame   Caller
00014F0E   10C10256   PrvCallWithNewStack+0014
000126AE   10CD2522   __Startup__+005C
0001268E   10CD3210   PilotMain+0032
00012674   10CD965E   EventLoop+0016
00012646   10C0EDD8   SysHandleEvent+03EC
00012572   10C4E206   Find+00A4
0001251A   10C47282   FrmDoDialog+0068
```

I dropped my Palm into the debugger using shortcut “.1” in the find dialog box. We can see exactly which functions are being called (assuming that they set their frame pointers correctly). Pretty cool. PalmDebugger knows the names of the functions because Metrowerks was set to compile these names into the code (for debugging purposes) as well as to generate frame pointers when it compiled the C code. Either option can be turned off if desired. The compiler has no problems generating code with no frame pointer.

Register saving

You probably noticed when you were using CS 110 library routines that they somehow managed to not trash any registers. The only registers they ever changed were ones that were used to return values back to you. How did they do that? As you can probably guess (especially after seeing our little trick with the frame pointer), they just saved the registers on the stack when they were called and then restored the registers when they were done. In other words, if we trash D1, D3, A0, A1, we could do:

```
MOVE.L     D1, -(A7)
MOVE.L     D2, -(A7)
MOVE.L     D3, -(A7)
MOVE.L     A0, -(A7)
MOVE.L     A1, -(A7)
...
...
MOVEA.L    (A7)+, A1
MOVEA.L    (A7)+, A0
MOVE.L     (A7)+, D3
MOVE.L     (A7)+, D2
MOVE.L     (A7)+, D1
```

...such a procedure is awfully annoying though. However, the 68K comes to the rescue again. This time, it's saving us with the `MOVEM` (move multiple) instruction. The `MOVEM` instruction

allows you to do what is known as “barfing” registers (also known less colorfully as “spilling” registers). We can write the above using MOVEM:

```
MOVEM.L    D1-D3/A0-A1, -(A7)
...
...
MOVEA.L    (A7)+, D1-D3/A0-A1
```

...neat-oh, huh? Now you know how we were able to present the nice abstraction to you in the CS 110 library.

Conventions for saving

Although in the CS 110 library routines we save every register, in the real world the abstraction is not so clean. In calling conventions, registers fall under two categories: caller saved and callee saved. Caller saved registers may be trashed by a called function. If a caller wants them to remain intact after a function call, he/she is responsible for saving them someplace. Callee saved registers, on the other hand, should not be trashed by a function call. If a called function wants to use those registers, it had better save them and then restore them later.

In Metrowerks Pascal and C calling conventions, registers D0-D2, and A0-A1 are caller saved. That means that if you call a function that is using C calling conventions, it is free to trash those registers. All other registers are callee saved. You are guaranteed that the caller won't touch them. Note that if you are not using Metrowerks C or Pascal calling conventions, you are free to make any given register callee or caller saved as you choose. The 68K itself couldn't care less.

C and Pascal conventions?

To finish up, I just want to say a quick word by what I mean by “C” and “Pascal” calling conventions. These calling conventions specify the way that functions written in C or Pascal (respectively) should call each other when running on PalmOS (which happens to have the same calling conventions as MacOS). They are quite arbitrary in the sense that other calling conventions could have been chosen that still would have worked. However, the reason that having a convention is important is that you want functions to have a well-defined boundary between them. This makes interoperability much simpler.

Note that the C and Pascal calling conventions that I've been talking about refer to the conventions for PalmOS on the 68K. Other operating systems/architectures may use completely different calling conventions. Other languages may use different calling conventions. Even C and Pascal programs can use different calling conventions if they don't need to interact with other programs.