# SUBPROGRAMS AND STACK

## SUBPROGRAMS

Higher level languages use subprograms all the time. Some languages have two types of subprograms: procedures (or subroutines) and functions, while others, like C, only have functions. Normally, the difference between the two is that functions return a value (only one!).
Assembly languages uses "functions" called "PROC"s and typically returns a value in AX register.
In C, we evoke functions by using the name with parentheses. In assembly language we put the instruction:

**call** *SubProg*

## Rules

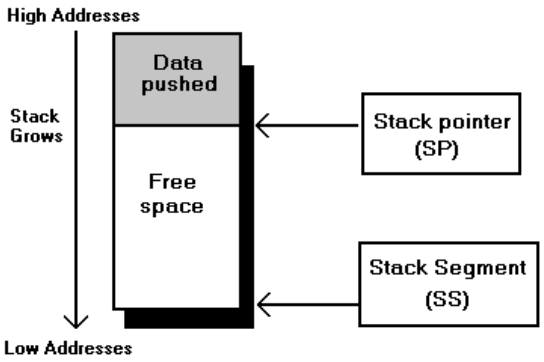
- Subprograms can be included in the same file or stored in a separate file.
- If they are in the same file, the ordering and naming is immaterial.
- There is no main(). The first PROC to be run is the one specified by END pseudo-op.
- Only the first PROC to execute the code to initialize the data segment register:

```
mov ax, @data
mov ds, ax
```

- For readability, put the .DATA before the .CODE for the subprogram:

```
            ...
Main        ENDP
            .DATA
            ...
            data for
            subprog
            ...
            .CODE
SubProg     PROC
            Code for
            subprog
```

- Unless labels are defined inside a PROC as PUBLIC, they are local to PROC that defined them.




## THE HARDWARE STACK

The hardware stack and stack pointer sp are necessary to get subprograms to work properly. Actually early CPUs did not have them, and as a result, they were more limited than what we have today. For instance, recursion was not possible. A stack is a data structure where data can only be added or removed at one end, so it is Last-In, First-Out (LIFO). There are special instructions built into the CPU to work with the stack.

The sp register points to the newest 16-bit value that is on the stack, which is the next item to be removed. You determine the size of the stack with the *.stack* directive. The assembler builds a program by putting the *.code* segment first, followed by the *.data* segment, and finally, the *.stack* segment. When putting an item on the stack, it starts at the highest address and grows down.

The instructions are:

- **push** (16-bit)
- **pushw** (16-bit)
- **pushd** (32-bit, 80386 and later)
- **pusha** (16-bit, push AX, CX, DX, BX, SP, BP, SI, DI, 80186 and later)
- **pushd** (32-bit, 80386 and later)
- **pushf** (16-bit, push the flag register)
- **pushfd** (32-bit, 80386 and later)
- **pop** (16- or 32-bit, based on the register specified, can not be sp register!)
- **popa** (16-bit, 80186 and later, pops in reverse order of pusha)
- **popad** (32-bit, 80386 and later)
- **popf** (16-bit)
- **popfd** (32-bit, 80386 and later)

Either a memory location (word or dword only, as appropriate), constant, or a register can be specified. If we have set up the data segment as:

```
X DW 1111
Y DW 2222
Z DW ?
```

We can have a stack that looks like:

```
?
?
?
```

In this case, the SP register really points to what is above the first location.

If we execute: push X
We now have a stack that looks like:

```
?
?
1111   <- SP
```

If we then execute: push Y
We now have a stack that looks like:

```
?
1111
2222   <-SP
```

If we finally execute: pop Z
We now have a stack that looks like:

```
?
?
1111  <- SP
```

The location for the variable Z is set to 2222 and the SP register points to the previous entry.
Note that stack location holding 2222 is considered unused and will be overwritten by the next push instruction. You can not count on items popped from the stack remaining in unused stack memory because the operating system also uses the stack.

The hardware stack is implemented as a normal block of memory of the size you specified in the .STACK pseudo-op. If you do push X followed by pop X, nothing is changed.
Saving and restoring registers is particularly important when using subprograms. You have the responsibility to save and restore important data in the registers before you call a subprogram and then you are responsible for restoring those registers afterwards.

***Subprograms should save and restore any registers that they use, unless they are returning values in certain registers!***

This means you have to write the instructions to do it!

***Subprograms should pop all items and only those items that they push on the stack!***

## CALL & RET

call and ret use the stack and the IP register. Remember the IP register always hold the address of the next instruction to be executed. The call instruction pushes the IP register contents onto the stack and then puts the address from the call instruction into the IP register.
Since that is the end of the call instruction, the computer then gets the first instruction from the subprogram and executes it. When the ret is reached (and it should be there!!! You have to put it there!), the contents of the last item pushed on the stack is put into the IP register and the SP register is adjusted to point to the next newest value. Since normally, what is copied into the IP register is the address of the instruction after the call that we were just discussing, the program continues from that point.

One of the most ingenious temporary uses of the stack is to call other subprogram. In fact a subprogram can call itself recursively, arbitrarily many times. The only thing to remember is the rule that a subprogram must pop off everything (and only those things) that it put onto the stack. That way, the return address will be available in the right position when the ret instruction is executed. When a subprogram is called within a subprogram, the call and corresponding ret take care to keeping the stack tidy automatically.

Continuing this ingenious use, the sub procedure can use the stack for its local variables. Remember in C, the variables declared in the body of the function exist only while the function is being executed. When the control passes back to the caller, the local functions disappear. What happens is that the local variables are created on the stack below the return address (we must store in a register what the starting address of the local variables is) and then we can refer to the local variables as an offset from that start. This way when we have recursion, each instance of the function can only see its version of the local variables! We will get to see this better when we talk about addressing mode and arrays.

## Separately Translating Subprograms

Putting subprograms into separate files lets you do things better and faster. Better, because you can use the subprograms in more than one program -- **Code reuse is good!** Faster because you only have to assemble those files with changes.

As an example, suppose we have the following file **main.asm**

```
;;          MAIN.ASM
INCLUDE  PCMAC.INC
         .MODEL      SMALL
         .STACK      100h
         .DATA
         PUBLIC      COMN
COMN     DW          ?
         ...
         .CODE
         EXTERN      SubProg : NEAR
Main     PROC
         mov         ax, @data
         mov         ds, ax
         call        SubProg
         call        SubProg
         _Exit
Main     ENDP
         END         Main
```

Then we need another file, let's call it **sub.asm**

```
;;          SUB.ASM
INCLUDE  PCMAC.INC                  ; if necessary
         .MODEL      SMALL
         .DATA                      ; if necessary for SubProg
         EXTRN       COMN : WORD
MSG      DB          'Hello', 10, 13, '$'
         .CODE
         PUBLIC      SubProg : NEAR
SubProg  PROC
         _PutStr     MSG
         ret
SubProg  ENDP
         END                        ; Can't have a label here
```

OK, now to assembly them together:

```
masm main
masm sub
link main + sub,,,util.lib;
```

or

```
ml main.asm sub.asm util.lib
```

Now if we change only sub.asm, we can do:

```
ml main.obj sub.asm util.lib
```

**Rules**

- Because assembly is done separately, each file that uses the macros must have the INCLUDE statement!
- If you call a subprogram that is in another file, you must have the EXTERN statement,
- Normally, only the main has the .STACK pseudo-op.
- For data that is defined in one file and used in another must have the EXTERN/PUBLIC pair, but notice that when you do that, you must provide the size (BYTE or WORD). **Normally, this is not a good way to do things because it creates a global variable. Use the stack instead if possible.**
- Only the main file has and END pseudo-op with a label.
- **EXTERN** can also be **EXTRN**.

## How the Linker Works

We have used the linker and have used library procedures, GetDec and PutDec. These have been assembled separately and stored in a library created by the author of our textbook. How does the linker handle that? When the assembler translates a source file into object code, it creates a symbol table of all the names and attributes of symbols defined in the file. When it is done, that symbol table is thrown away. Since PUBLIC symbols may be defined in one file and referenced in another file(s), the assemblers saves two files in the .obj file -- a table of EXTRN symbols and a list of places where each symbol is referenced, and a table of PUBLIC symbols and the unique place where each is defined.
The unresolved external references must be resolved by the linker. Once the linker knows where one of the PUBLIC is symbols is stored, it goes back and modifies the locations of the EXTRN references with the now known address.