# Experiment 6: Stack Operations, Subroutines, and Memory

**Instructional Objectives:**

- To understand the way that the stack is implemented to save register values.
- To use the stack to pass information to and from a subroutine.
- To understand the way that banked memory and paging is implemented in the HC(S)12 microcontroller

**Introduction:**

Previous lab procedures have required that students use the HC(S)12 instruction set to write code to interface with the lab I/O board. However, most programming is not done in assembly because of its lack of portability from one processor to another. One drawback of using higher level languages is not being able to observe the several uses of the stack. The purpose of the following experiment is to give students a better appreciation for the tasks the stack fulfills in a program, as well as a better understanding of the way that memory is structured in the HC(S)12 used in lab.

**Stack Manipulation:**

The stack is a Last In First Out data structure that programmers use to temporarily store and retrieve data. In previous experiments, the stack has been implemented in short delay subroutines, but it is important to understand how to manipulate the stack, as well as what it's functions are in a programming environment.

**Note:** Remember that the way to properly initialize the stack pointer for the s12e128 used in lab is to use the constant `__SEG_END_SSTACK` that is referenced at the top of a new program. This is a value that is generated by the linker file. Its value is whatever the starting address of RAM is plus the value of `STACKSIZE`. On default, this value is `0x0500`, but this value can be changed to increment or decrement the size of the stack by changing the value of `STACKSIZE` in the linker file.

On the next page, the several different uses of the stack will be explored in depth. The stack has five primary functions in the programming environment:

- Saving the return address for properly returning from a subroutine or a function.
- Temporarily saving registers for a subroutine or a function.
- Saving local variables used in a subroutine or function.
- Passing function parameters to a subroutine or function.
- Returning data from a subroutine or function

The next couple of pages have some examples of how the stack is implemented.

- **Preserving the return address**

  Consider the following code to call the delay subroutine and the accompanying stack frame:

```
Memory      Calling
Location    Program

C000:       STAA   PTU
C003:       JSR    delay
C006:       LDAA   PTU
```

| | Stack Address | Byte Data |
|---|---|---|
| | $4FD | |
| SP | $4FE | $C0 |
| | $4FF | $06 |
| SP+2 | | |

  Above is a sample portion of code from the keypad from experiment 5.2. Suppose that the stack pointer currently holds the value `$500`. When the instruction `JSR delay` is executed, the program **pushes** the contents of the program counter onto the stack, and the stack pointer is automatically decremented by two so that it now holds the value `$4FE`. Later at the end of the subroutine, when the `RTS` instruction is executed (see below), the program **pulls** the data that the stack pointer is pointing to from the stack, and the stack pointer is automatically incremented by two.

- **Temporarily saving register values**

  Continuing with the example above, consider the following subroutine and accompanying stack frame:

```
delay:  PSHX
        LDX     delaycount
lp:     DEX
        BNE     lp
        PULX
        RTS
```

| | Stack Address | Byte Data |
|---|---|---|
| SP | $4FC | XH |
| | $4FD | XL |
| SP+2 | $4FE | $C0 |
| | $4FF | $06 |
| SP+4 | | |

  The above example is the delay loop that was implemented in lab 4 for the stepper motor. Since index register X is being used in the subroutine, at the beginning of the subroutine, the contents of index register X is **pushed** on the stack, and at the end of the subroutine, the contents of index register X is **pulled** from the stack. Since the HC(S)12 only has a limited number of registers for use, it is important to preserve the contents of any registers that are used in a subroutine on the stack.

- **Passing arguments to a subroutine**

The following is an excerpt of the code from lab 6.2. Please note the accompanying assembly code and stack frame.

```
SendMessageInfo(Data,numberChars, dummy);

388026 LEAX   0,SP
388028 PSHX
388029 LDD    8,SP
38802B PSHD
38802C LDD    12,SP
38802E CALL   0x8074,0x38
```

| | Stack Address | Byte Data |
|---|---|---|
| SP → | $4ED | Page |
| | $4EE | Return Address High |
| | $4EF | Return Address Low |
| SP+3 → | $4F0 | 2nd Arg Low |
| | $4F1 | 2nd Arg Low |
| SP+5 → | $4F2 | 1st Arg High |
| | $4F3 | 1st Arg Low |

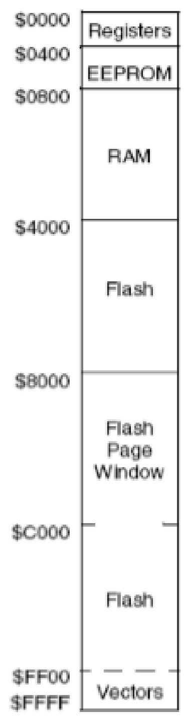
The above function call is compiled down to six assembly instructions. This function accepts three arguments. The first is the two-byte effective address of the character array `Data`. The second argument is the two-byte integer value `numberChars` which holds the number of characters in the array. The third argument is a dummy argument whose sole purpose is to suppress a compiler warning. The first two arguments are passed to the subroutine through the stack. The third argument remains in accumulator D. **For any function call using the default compiler settings in CodeWarrior, all of the arguments except for the last argument are passed through the stack.** The program then uses the `CALL` instruction to preserve the return address and page number and jump to the subroutine.

**Extended Memory:**

One new thing to take away from the above example is the use of the instruction `CALL` instead of `JSR` to call the function. It's important to understand the memory model of the HC(S)12 and distinction between banked and non-banked memory. Remember that the number of addressable bytes with a 16-bit data bus is $2^{16} = 65{,}536$ bytes. However, the s12e128 used in the lab has 128k of flash memory available for use. To access these addition locations, a technique called "paging" is implemented.

**Figure 2. CPU Local Map for an HCS12 Device**




The above figure shows the local memory map for an HC(S)12 device. There are two common characteristics for devices in the HC(S)12 family. The first is that the space boundaries of register space, EEPROM, and RAM may differ between devices. The other is that the lower 48kB hosts the flash memory. On the HC(S)12, flash memory is partitioned into 16kB sections. The middle 16kB region is called the flash page window.

Certain local addresses do not always point to the same physical address location. These special address ranges are known as page windows. Local addresses in these ranges are addresses of 16 bits and do not have enough information for the controller to determine what the physical location of this memory space is. This addition information is stored to the `PPAGE` register, which is used to select the part of physical memory that the page window points to.

```
/* non-paged FLASHs */
      ROM_4000      = READ_ONLY     0x4000 TO   0x7FFF;
      ROM_C000      = READ_ONLY     0xC000 TO   0xFEFF;
 /*   VECTORS       = READ_ONLY     0xFF00 TO   0xFFFF; intentionally not defined: used for VEC
   //OSVECTORS      = READ_ONLY     0xFF88 TO   0xFFFF;   /* OSEK interrupt vectors (use your v

/* paged FLASH:                     0x8000 TO   0xBFFF; addressed through PPAGE */
      PAGE_38       = READ_ONLY   0x388000 TO 0x38BFFF;
      PAGE_39       = READ_ONLY   0x398000 TO 0x39BFFF;
      PAGE_3A       = READ_ONLY   0x3A8000 TO 0x3ABFFF;
      PAGE_3B       = READ_ONLY   0x3B8000 TO 0x3BBFFF;
      PAGE_3C       = READ_ONLY   0x3C8000 TO 0x3CBFFF;
      PAGE_3D       = READ_ONLY   0x3D8000 TO 0x3DBFFF;
/*    PAGE_3E       = READ_ONLY   0x3E8000 TO 0x3EBFFF; not used: equivalent to ROM_4000 */
/*    PAGE_3F       = READ_ONLY   0x3F8000 TO 0x3FBEFF; not used: equivalent to ROM_C000 */
END
```

The above was taken from the `linker.prm` file in a default CodeWarrior project. Under paged flash, the first byte is the information stored in the PPAGE register and the second and third byte is the local address. The HC(S)12 family chooses pages in sequential order in such a way that the last page of memory is given the number `0x3F`. Although the concept of paging and page numbers are only useful when accessing banked locations, it is important to understand that the division of numbered pages is done for all of memory. For more information on the memory scheme in the S12 architecture, visit the following resource from NXP: http://www.nxp.com/assets/documents/data/en/application-notes/AN3784.pdf

The reason that the memory scheme is brought up is because most c projects in CodeWarrior, including project 6.2 and the final project, implement this page window. In order to preserve the physical page when jumping to a subroutine, the `CALL` instruction is used in place of the `JSR` instruction. `CALL` preserves the page in the `PAGE` register as well as the return address. To return from `CALL`, the `RTC` (Return from call) instruction should be used instead of the `RTS` instruction, or else the program will not pull the correct value from the `PPAGE` register and thus will not return properly.

**Experimental Procedure:**

**Laboratory 6.1: Stack Manipulation**

Make a new program. At the beginning of the program, initialize the stack pointer using the `__SEG_END_SSTACK` variable generated by the linker file. Load the registers with the following values:

- Accumulator A = `$11`
- Accumulator B = `$22`
- Index Register X = `$3333`
- Index Register Y = `$4444`

After the registers have been loaded with the values above, call a subroutine. In this subroutine, push the registers onto the stack in the order that they are loaded above. Clear all the registers. Pull the values back from the stack in the opposite order they were pulled.

When running the program in the debugger, go to the location of the stack pointer in the memory window and single step through the program. Fill out the corresponding stack frame below.

| Stack Address | Byte Data |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

**Laboratory 6.2: Passing Parameters**

The objective of this exercise is to write an assembly language subroutine to interface with a C language main program and a C language subroutine. The subroutine to be written is:

```
void SendMessageInfo(char* Data, int numberChars, int dummy)
```

The above function call passes three 16-bit arguments to the subroutine `SendMessageInfo`. The first argument is the starting effective address of a character string. The second argument is the number of characters in the string. The third argument is a dummy argument to suppress a compiler warning that can be ignored.

The first and second arguments are passed to the function through the stack. The third argument remains in accumulator D. Since this program uses banked memory, both the return address and page number are saved on the stack via the CALL instruction. Structure of stack after the CALL instruction is shown as below:

| **Stack Address** | **Byte Data** |
|---|---|
| SP | Page |
| SP+1 | Return Address High |
| SP+2 | Return Address Low |
| SP+3 | Number of Character High |
| SP+4 | Number of Character Low |
| SP+5 | Array Starting Effective Address High |
| SP+6 | Array Starting Effective Address Low |

Download and extract the file `Lab6-2` from the lab section of the course site. Create a new assembly file and add it to the project. Call the subroutine `SendMessageInfo`. This subroutine should do the following:

- Load the effective address of array named `Data` from the stack into one of the index registers. Load the counter(number of characters) into one of the remaining 16-bit registers.
- Load the 8-bit values from `Data` one at a time.
- Push the argument through the stack to the function `void SendsChr(char OutValue, int Dummy)`. This function accepts two arguments. The first is the 8-bit value from the array. The second is a dummy argument passed through accumulator D and can be ignored.
- Use the `CALL` instruction to call the `SendsChr` function.
- Adjust the stack pointer so that the program returns from `SendMessageInfo` properly.
- Decrement and check the value of `numberChars`. If the count has not reached zero, branch back and load the next element of `Data`. Else, use the `RTC` instruction to properly return from the array.

- **Remember** that SendMessageInfo and SendsChr both must be defined and referenced using the appropriate XDEF and XREF assembler directives.

EXTRA INFO: In assembly, SendMessageInfo function call is compiled down into the following instructions:

```
LEAX    0,SP
PSHX
LDD     8,SP
PSHD
LDD     12,SP
CALL    0X8074,0x38
```

**Laboratory 6.3: LCD and Potentiometer**

The object of this lab is to create an assembly program that reads the values from the potentiometer on the I/O board and display them on the LCD. Extract and open the file `Lab6-3` from the lab section of the course site.

To display a string on the LCD, take the following steps (the first two are already completed in the file found in the course site):

- Create a string that contains 33 characters in the variables section. To indicate the end of the string, use a null character (0) for the terminator.
- Initialize the characters of the string one at a time in the code section using the `MOVB` instruction.
- Initialize the LCD by calling the `init_LCD` function. This is a function of type void that takes no arguments. Since this project does not implement banked memory, use the `JSR` instruction to call this function. This should only be done once at the start of the program. Doing this multiple times will cause the LCD to flicker.
- To display a string on the LCD, load the effective address of the string into accumulator D, then call the `display_string` function to display the string on the LCD. This is a function of type void and takes the effective address of a char array as it's only argument. Reaching a null character will terminate this function. Repeat this step any time that a change is made to the string.

> Note: The LCD utilizes the ASCII table to display characters. To change an individual character in a string, put the new character in the corresponding memory address of that string. The value of the new character must be in ASCII format.

To read a value from the potentiometer, call the Read_Pot function. This is a function of type void that takes no arguments. After this function is called, the value of the potentiometer is stored into the two byte variable pot_value. The value of the potentiometer is only one-byte, and is in the lower byte of pot_value. This value can take the range from 0 to 255, but on the lab I/O board it generally only goes up to around 127.

To complete this exercise, take the following steps (note that steps 1 and 3 are already done in the `Lab6-3` file):

- Include the following files in the project:
  - `funct.h` (Header file for the LCD and Potentiometer)
  - `lcddisp.c` (C code for `init_LCD` and `display_string`)
  - `potentiometer.c` (C code for the `read_pot` and `pot_value`)
- Reference the functions and variables that are going to be used in the program.
- Create the string that will be displayed to the LCD. The string should be:
  - `"The value of the pot is:        ",0`
- Initialize the LCD using the `init_LCD` function.
- Get the value from the potentiometer using the `read_pot` function.
- Store the value of the potentiometer in the string. The value of the potentiometer is from 0 to 255. To do so, follow the following three steps:
  - Separate the digits. For example, if the value in the potentiometer is 147:
    - 147/100: Quotient = 1, Remainder = 47. Store the quotient into memory.
    - 47/10: Quotient = 4, Remainder = 7 (47 is the remainder from 147/100). Store the quotient and remainder into memory.
  - Add #$30 to each digit to convert to its ASCII equivalent.
  - Store the hundreds, tens, and ones digit in order into the string after the ':' character.
- Display the string to the LCD using the `display_string` function.
- Repeat the previous three steps.