**Austin Sypolt**

**ECE 362**

**Pre-Lab #6 Stack Operation, Subroutines and Memory**

**Introduction:**

The purpose of this lab is to properly understand how to use the stack in addition to the various applications of using a stack while programming projects. Additionally, we will attempt to understand how banked memory and paging is implemented in our microcontroller.

**Laboratory 6.1: Stack Manipulation**

Objective/Purpose:

The purpose of this program is to load registers with specified values, then push them into the stack in properly loaded order. Then all the registers should be cleared and the values from the stack should be pulled back in the opposite order they were pushed.

Expected Results:  
Post execution all the registers should have a value loaded back into them however the values should be reversed.

A=$44, B=$44, X=$3333, Y=$4444

Code:

XDEF Entry

XREF \_\_SEG\_END\_SSTACK

**Lab 6.2: Passing Parameters**

Objective/Purpose:

The purpose of this lab section is to make an assembly language subroutine that interfaces with a C language program and a C subroutine.

Expected Results:

It is expected that both subroutines run through without error with proper utilization of the stack in both assembly and C.

Code:

XDEF Entry

XREF \_\_SEG\_END\_SSTACK

**Lab 6.3: LCD and Potentiometer**

Objective/Purpose:

The purpose of this is to make an assembly program to read values from a potentiometer and then display them on the LCD.

Expected Results:

Expecting for the switch inputs to send the proper values to the LEDs.

Code:

XDEF Entry

XREF \_\_SEG\_END\_SSTACK

ECE362 Lab 6 Pre-lab

Austin Sypolt

10/10/18

**Introduction:**

Understand the way the stack is implemented to save register values

Use the stack to pass information to and from a subroutine

Understand the way that banked memory and paging is implemented in the HC(S)12 microcontroller

Previously we used the HC(S)12 instruction set to write code to interface with the lab I/O. However, most programming is not done in assembly because of its lack of portability from one processor to another. Drawbacks from using higher level language is not being able to observe the several uses of the stack.

This lab focuses on teaching students to properly understand and apply the way memory is structured in the HC(S)12 used in lab.

**Stack Manipulation:ss**

Stack acts as LIFO system (last in first out), this is used to temporarily store and retrieve data.

In previously labs we have used the stack to implement short delay subroutines, but we’re going to focus on how to manipulate the stack through functions in a programming environment.

To initialize the stack pointer for the s12e128 used in lab:

\_\_SEG\_\_END\_SSTACK – This is referenced at the top of a new program

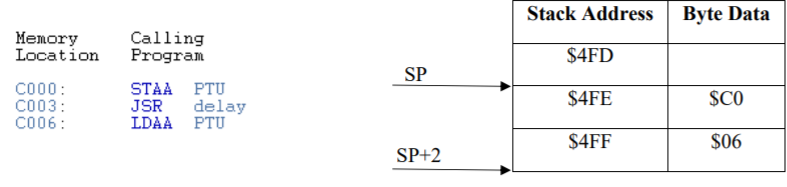
This is a value generated by the linker file. It is whatever the starting address of RAM is plus the value of STACKSIZE. On default, this value is 0x0500, but this value can be changed to increment and decrement the size of the stack by hanging the value of STACKSZE in the linker file.

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
* Returning data from a subroutine or function

**How a stack is implemented:**

Code to call the delay subroutine and accompanying stack frame:



This is a sample of code from the keypad from experiment 5.2.

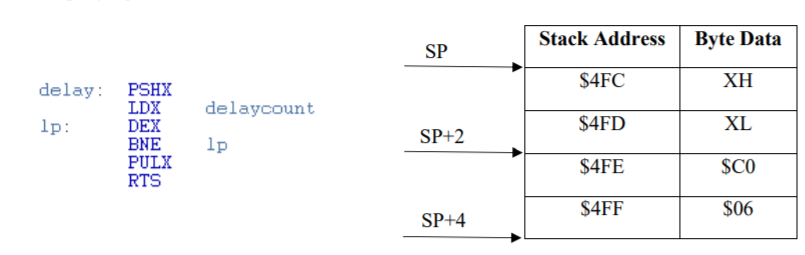
Suppose the stack pointer holds the value $500.

When JSR delay is executed the program **pushes** the contents of the program counter onto the stack, and the stack pointer automatically decrements by 2. Now it will hold the value $4FE ($500 – 2 = $4FE).

At the end of the subroutine, when RTS is executed, the program **pulls** the data that the stack pointer is pointing to **from** the stack. The stack pointer is then automatically incremented by 2.

**Temporarily Saving Register Values**

Extended examples from (**How a stack is implemented**):

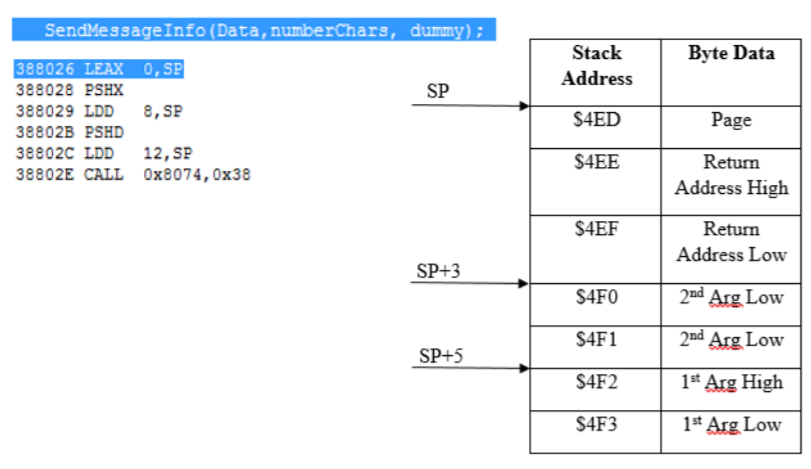


This is a sample of code that was implemented in lab 4 for the stepper motor.

Since index register X is used in the subroutine, the contents of index register X is **pushed** on the stack (at the beginning of the subroutine). Then the contents of the index register X is **pulled** from the stack (at the end of the subroutine). **Since the HC(S)12 only has a limited number of registers for use, is very important to preserve the contents of any registers that are used in a subroutine on the stack.**

**Passing Arguments to a Subroutine**

This sample of code you will be using from lab 6.2



In this function call there are 6 assembly instructions. The function accepts three arguments.

First: Two-byte effective address of the character array Data

Second: Two-byte integer value numberChars which holds the number of characters in the array

Third: Dummy argument whose sole purpose is to suppress a compiler warning

The first two arguments are passed to the subroutine through the stack, while 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.**

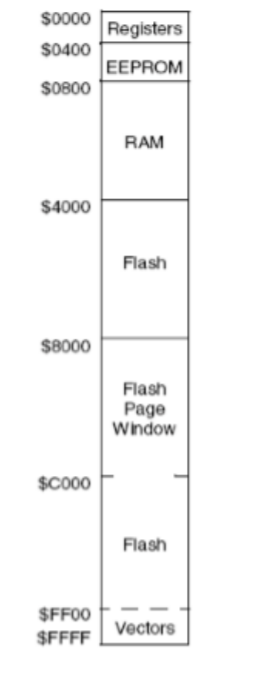
The program then uses the CALL instruction to preserve the return address and page number and jump to the subroutine.

**Extended Memory:**

In the above examples we use CALL instead of JSR to call the function.

There is a distinction between **banked** and **non-banked** memory.

The number of addressable bytes with a 16-bit data bus is 2^16 = 65536, however the se12e128 used in lab has 128k of flash memory available for use. In order to access these addition locations we use **paging**.



This is the local memory map from an HC(S)12 device.

There are two common characteristics for devices in the HC(S) family.

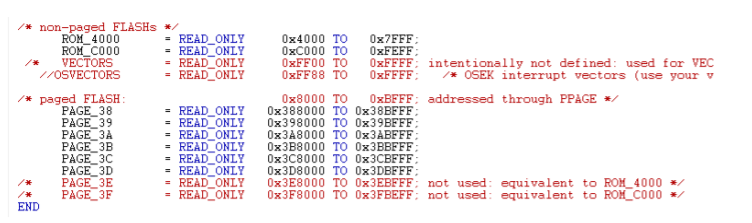
1. The space boundaries of register space, EEPROM, and RAM may different between devices
2. The lower 48kB hosts the flash memory

On the HC(S)12, **flash memory** is partitioned into 16kB sections. The middle region of the 16kB is the flash page window.

Sometimes a local address does 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 carry enough information for the controller to determine what the physical location of this memory space is.

This addition information however is storage to the PPAGE register, and it is used to select the part of physical memory that the page window points to.

**Extended Memory (Continued):**

This was taken from the linker.prm file in a default CodeWarrior project.

Under ‘/\*paged Flash:’, the first byte is the 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 if given the number 0x3F.

Paging and page numbers are only useful when accessing banked locations, it is still useful to understand that the division of numbered pages is done for all of memory.

This memory scheme is brought up as **most** c projects in CodeWarrior (including 6.2 and the final project) use this page window.

CALL preserves the page in the PAGE register as well as the return address

To return from CALL, the RTC (Return from call) instruction is used instead of RTS, otherwise the program does not pull the correct value from the PPAGE register.