The stack and the stack pointer

If you “google” the word stack, one of the definitions you will get is:

A reserved area of memory used to keep track of a program’s internal operations, including functions, return addresses, passed parameters, etc. A stack is usually maintained as a “last in, first out” (LIFO) data structure, so that the last item added to the structure is the first item used.

Sometimes is useful to have a region of memory for temporary storage, which does not have to be allocated as named variables.

When you use subroutines and interrupts it will be essential to have such a storage region.

Such region is called a **Stack**

The **Stack Pointer** (SP) register is used to indicate the location of the last item put onto the stack.

When you PUT something ONTO the stack (PUSH onto the stack), the SP is decremented before the item is placed on the stack.

When you take something OFF of the stack (PULL from the stack), the SP is incremented after the item is pulled from the stack.

Before you can use a stack you have to initialize the SP to point to one value higher than the highest memory location in the stack.

For the HC12 use a block of memory from about $3B00 to $3BFF for the stack.

For this region of memory, initialize the stack pointer to $3C00. Use LDS (Load Stack Pointer) to initialize the stack pointer.

The stack pointer is initialized only one time in the program.
The stack is an array of memory dedicated to temporary storage

SP points to location last item placed in block

**SP decreases** when you put an item on the stack

**SP increases** when you pull the item from the stack

For the HC12, use **0x3c00** as initial SP

```
STACK:
EQU        $3C00
LDS #STACK
```
Stack pointer:

Initialize ONCE before the first use (LDS #STACK)

Points to last used storage location

Decreases when you put something on stack, and increases when you take something off stack

STACK:       equ $3C00
             lds #STACK
             ldax #$2e
             ldx #$1254
             psha
             pshx
             clra
             ldx #$fff

CODE THAT USES A & X

pulx
pula
An example of some code which uses the stack

**PSHA**

Push A onto Stack

**Operation**

\[(SP) \rightarrow (SP) \quad \text{SP} \downarrow 1 \]

\[(A) \rightarrow M_{SP} \]

Decrement SP by one and loads the value in A into the address to which SP points.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

**CCR Effects**

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>H</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Code and CPU Cycles**

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Machine Code (Hex)</th>
<th>CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSHA</td>
<td>INH</td>
<td>3E</td>
<td>0E</td>
</tr>
</tbody>
</table>

**PSHX**

Push X onto Stack

**Operation**

\[(SP) \rightarrow (SP) \quad \text{SP} \downarrow 2 \]

\[(X_9) \quad (X_{0}) \rightarrow M_{SP} \quad M_{SP+1} \]

Decrement SP by two and loads the high byte of X into the address to which SP points.

Loads the low byte of X into the address to which SP points plus one. After PSHX executes, SP points to the stacked value of the high byte of X.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

**CCR Effects**

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>H</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Code and CPU Cycles**

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Machine Code (Hex)</th>
<th>CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSHX</td>
<td>INH</td>
<td>34</td>
<td>0E</td>
</tr>
</tbody>
</table>
Subroutines

A subroutine is a section of code which performs a specific task, usually a task which needs to be executed by different parts of the program.

Example:

```
org $1000
:
:
call sqrt
  ??
call sqrt
:
swi
compute square root
:

jmp label
```

-Math functions, such as square root (sqrt)

Because a subroutine can be called from different places in a program, you cannot get out of a subroutine with an instruction such as

```
jmp label
```

Because you would need to jump to different places depending upon which section of the code called the subroutine.

When you want to call the subroutine your code has to save the address where the subroutine should return to. It does this by saving the return address on the stack.

- This is done automatically for you when you get to the subroutine by using JSR (Jump to Subroutine) or BSR (Branch to Subroutine) instruction. This instruction pushes the address of the instruction following the JSR (BSR) instruction on the stack.

After the subroutine is done executing its code, it needs to return to the address saved on the stack.

- This is done automatically when you return from the subroutine by using RTS (Return from Subroutine) instruction. This instruction pulls the return address off the stack and loads it into the PC.
Subroutines

Caution: The subroutine will probably need to use some HC12 registers to do its work. However, the calling code may be using its registers for some reason – the calling code may not work correctly if the subroutine changes the values of the HC12 registers.

To avoid this problem, the subroutine should save the HC12 registers before it uses them, and restore the HC12 registers after it is done with them.
Example of a subroutine to delay for certain amount of time

; Subroutine to wait for 100 ms

Delay: ldaa #250
Loop2: ldx #800
Loop1: dex
       bne Loop1
       deca
       bne Loop2
       rts

What is the problem with this subroutine?

It changes the values of the registers that are most frequently used: A and X

How can we solve this problem?
Example of a subroutine to delay for certain amount of time

To solve, save the values of A and X on the stack before using them, and restore them before returning.

; Subroutine to wait for 100 ms

<table>
<thead>
<tr>
<th>Delay:</th>
<th>psha</th>
<th>pshx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ldaa</td>
<td>#250</td>
</tr>
<tr>
<td>Loop2:</td>
<td>ldx</td>
<td>#800</td>
</tr>
<tr>
<td>Loop1:</td>
<td>dex</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>Loop1</td>
</tr>
<tr>
<td></td>
<td>deca</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>Loop2</td>
</tr>
<tr>
<td></td>
<td>pulx</td>
<td>; restore registers</td>
</tr>
<tr>
<td></td>
<td>pula</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rts</td>
<td></td>
</tr>
</tbody>
</table>
A sample program

; Program to make binary counter on LEDs
; The program uses a subroutine to insert a delay between counts

prog:     equ     $1000
STACK:    equ     $3C00
PORTA:    equ     $0000
PORTB:    equ     $0001
DDRA:     equ     $0002
DDRB:     equ     $0003

org prog

lds #STACK ; initialize stack
ldaa #$ff ; put all 1s into DDRA
staa DDRA ; to make PORTA output
clr PORTA ; put $00 into PORTA

loop:     jsr delay ; wait a bit
inc PORTA ; add 1 to PORTA
bra loop   ; repeat forever

; Subroutine to wait for 100 ms

delay:    psha
          pshx
          ldaa #250
loop2:    ldx #800
loop1:    dex
          bne loop1
          deca
          bne loop2
          pulx
          pula
          rts
JSR and BSR place return address on stack
RTS returns to instruction after JSR or BSR

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 CF 3C 00</td>
<td>equ</td>
<td>$3C00</td>
</tr>
<tr>
<td>1003 16 10 07</td>
<td>org</td>
<td>$1000</td>
</tr>
<tr>
<td>1006 7F</td>
<td>lds</td>
<td>#STACK</td>
</tr>
<tr>
<td>1007 CE 12 34</td>
<td>jsr</td>
<td>MY_SUB</td>
</tr>
<tr>
<td>100A 3D</td>
<td>swi</td>
<td></td>
</tr>
<tr>
<td>1003 16 10 07</td>
<td>jsr</td>
<td>MY_SUB</td>
</tr>
<tr>
<td>1006 7F</td>
<td>swi</td>
<td></td>
</tr>
<tr>
<td>1007 CE 12 34</td>
<td>jsr</td>
<td>MY_SUB</td>
</tr>
<tr>
<td>100A 3D</td>
<td>swi</td>
<td></td>
</tr>
</tbody>
</table>

Memory used by MCU:
- 0x3BF6
- 0x3BF7
- 0x3BF8
- 0x3BF9
- 0x3BFA
- 0x3BFB
- 0x3BFC
- 0x3BFD
- 0x3BFE
- 0x3BFF
- 0x3C00

A
B
D
X
Y
SP
PC
CCR
Another example using a subroutine

Using a subroutine to wait for an event to occur then take action

Wait until bit 7 of address $00CC is set.

Write the value of ACCA to address $00CF

; This routine waits until the HC12 serial port is ready, then send a byte of data to the serial port

putchar:  brclr  $00CC,#$80,putchar ; Data Terminal Equip. ready
           staa $00CF  ; Send char
           rts

; Program to send the word “hello” to the HC12 serial port

loop:     ldx #str
          ldaa 1,x+
          beq done
          jsr putchar
          bra loop
done:     swi

str:      fcc “hello” ; form constant character
          dc.b $0a,$0d,0 ; CR-LF
Another example using a subroutine

A complete program to write to the screen

prog: equ $1000
data: equ $2000
stack: equ $3c00

org prog
lds #stack
ldx #str
loop: ld aa 1,x+
beq done
jsr putchar
bra loop

character
done: swi

putchar: brclr $00CC,$80,putchar
ready
staa $00CF
rts

str: fcc “hello”
dc.b $0a,$0d,0

; initialize stack
; load pointer to “hello”
; is done then end program
; write character to screen
; branch to read next
; check is serial port is ready
; and send
; form constant character
; CR-LF
JSR and BSR place return address on stack
RTS returns to instruction after JSR or BSR

---

**JSR**

**Jump to Subroutine**

**Operation**

- $(SP) - 5(002) \rightarrow SP$
- $(RTN) \rightarrow (MSP)(MSP + 1)$

Subroutine address \( \rightarrow PC \)

- Sets up conditions to return to normal program flow, then transfers control to a subroutine.
- Uses the address of the instruction following the JSR as a return address.
- Decrement SP by two, to allow the two bytes of the return address to be stacked.
- Stacks the return address (SP points to the high byte of the return address).
- Calculates an effective address according to the rules for extended, direct, or indexed addressing.

Subroutines are normally terminated with an RTS instruction, which restores the return address from the stack.

---

**CCR Effects**

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>H</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

---

**Code and CPU Cycles**

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Machine Code (Hex)</th>
<th>CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR opr3</td>
<td>DSR</td>
<td>17 dd</td>
<td>SPPP</td>
</tr>
<tr>
<td>JSR opr15</td>
<td>EXT</td>
<td>15 th 11</td>
<td>SPPP</td>
</tr>
<tr>
<td>JSR oprQ4</td>
<td>IDX0</td>
<td>15 xb</td>
<td>PPPS</td>
</tr>
<tr>
<td>JSR opr5_4y &amp; sps</td>
<td>IDX1</td>
<td>15 xb ff</td>
<td>PPPS</td>
</tr>
<tr>
<td>JSR opr5_4y &amp; sps</td>
<td>IDX2</td>
<td>15 xb ee ff</td>
<td>PPPS</td>
</tr>
<tr>
<td>JSR opr16 &amp; sps</td>
<td>[IDX]</td>
<td>15 xb ee ff</td>
<td>PPPS</td>
</tr>
<tr>
<td>JSR opr16 &amp; sps</td>
<td>IDX2</td>
<td>15 xb ee ff</td>
<td>PPPS</td>
</tr>
</tbody>
</table>
Using DIP switches to get data into the HC12

DIP switches make or break a connection (usually to ground)

---

5V
Using DIP switches to get data into the HC12

To use DIP switches, connect one end of each switch to a resistor

Connect the other end of the resistor to +5V

Connect the junction of the DIP switch and the resistor to an input port on the HC12

When the switch is open, the input port sees a logic 1 (+5V)

When the switch is closed, the input sees a logic 0 (0V)
Looking at the state of a few input pins

Want to look for a particular pattern on 4 input pins

-For example want to do something if pattern on PB3-PB0 is 0110

Don't know or care what are on the other 4 pins (PB7-PB4)

Here is the wrong way to do it:

```
ldaa PORTB
cmpa #b0110
beq task
```

If PB7-PB4 are anything other than 0000, you will not execute the task.

You need to mask out the Don’t Care bits before checking for the pattern on the bits you are interested in

```
ldaa PORTB
anda #b00001111
cmpa #b00000110
beq task
```

Now, whatever pattern appears on PB7-4 is ignored
Using an HC12 output port to control an LED

Connect an output port from the HC12 to an LED.

Using an output port to control an LED

PA0

Resistor, LED, and Ground connected internally inside breadboard

When a current flows Through an LED, it emits light
Making a pattern on a 7-segment LED

Want to make a particular pattern on a 7-segment LED.

Determine a number (hex or binary) that will generate each element of the pattern

- For example, to display a 0, turn on segments a, b, c, d, e, and f, or bits 0, 1, 2, 3, 4, and 5 of PTH. The binary pattern is 00111111, or $3f$
- To display 0, 2, 4, 6, 8, the hex numbers are $3f$, $5b$, $66$, $7d$, $7f$.

Put the numbers in a table

Go through the table one by one to display the pattern

When you get to the last element repeat the loop
Flow chart to display the patterns on a 7-segment LED

start

Port H Output

Point to First entry

Get entry

Output to PORT H

Inc pointer

L1:

ldaa #$ff
sta DDRH

ldx #table

ldaa 0,x

sta PORTH

inx

cpx #end_table

bres L2

bra L1

L2:

Idxx #table

ldaa 0,x

Idaa PORTH

inx

cpx #end_table

bres L2

bra L1

X < end
Program to display the patterns on a 7-segment LED

; Program to display patterns

prog:    equ $1000
data:    equ $2000
stack:   equ $3C00
PTH:     equ $0260
DDRH:    equ $0262
org      equ prog
lds      #stack
ldaa     #$ff
staa     DDRH
L1:      ldx #table
L2:      ldaa 1,x+
staa     PTH
jsr      delay
cpx      #table_end
bls      L2
bra      L1
delay:   psha
          pshx
          ldaa #250
Loop2:   ldx #8000
Loop1:   dex
          bne  Loop1
          deca
          bne  Loop2
          pulx
L1:      ldx #table
L2:      ldaa 1,x+
staa     PTH
jsr      delay
cpx      #table_end
bls      L2
bra      L1
table:   dc.b $3f
          dc.b $5b
          dc.b $66
          dc.b $7d
          org  data
table_end: dc.b $7f