- **HC12 Addressing Modes**
  - Inherent, Extended, Direct, Immediate, Indexed, and Relative Modes
  - Summary of MC9S12 Addressing Modes
  - Using X and Y registers as pointers
  - How to tell which branch instruction to use

- **Instruction coding and execution**
  - How to hand assemble a program
  - Number of cycles and time taken to execute an MC9S12 program

The MC9S12 has 6 addressing modes

Most of the HC12's instructions access data in memory
There are several ways for the HC12 to determine which address to access

Effective address:
Memory address used by instruction (all modes except INH)

Addressing mode:
How the MC9S12 calculates the effective address

HC12 ADDRESSING MODES:

INH Inherent

IMM Immediate

DIR Direct

EXT Extended

REL Relative (used only with branch instructions)

IDX Indexed (won't study indirect indexed mode)

The Inherent (INH) addressing mode

Instructions which work only with registers inside ALU

ABA ; Add B to A (A) + (B) → A
18 06

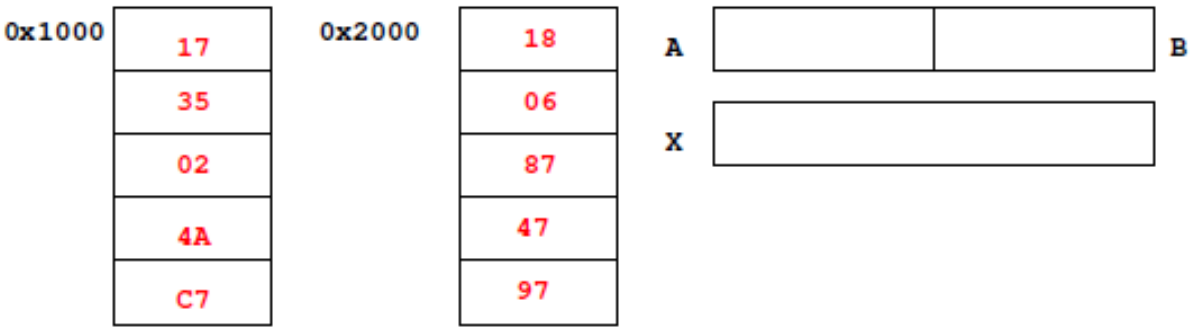
CLRA ; Clear A 0 → A
87

ASRA ; Arithmetic Shift Right A
47

TSTA ; Test A (A) − 0x00 Set CCR
97

The HC12 does not access memory

There is no effective address



The Extended (EXT) addressing mode

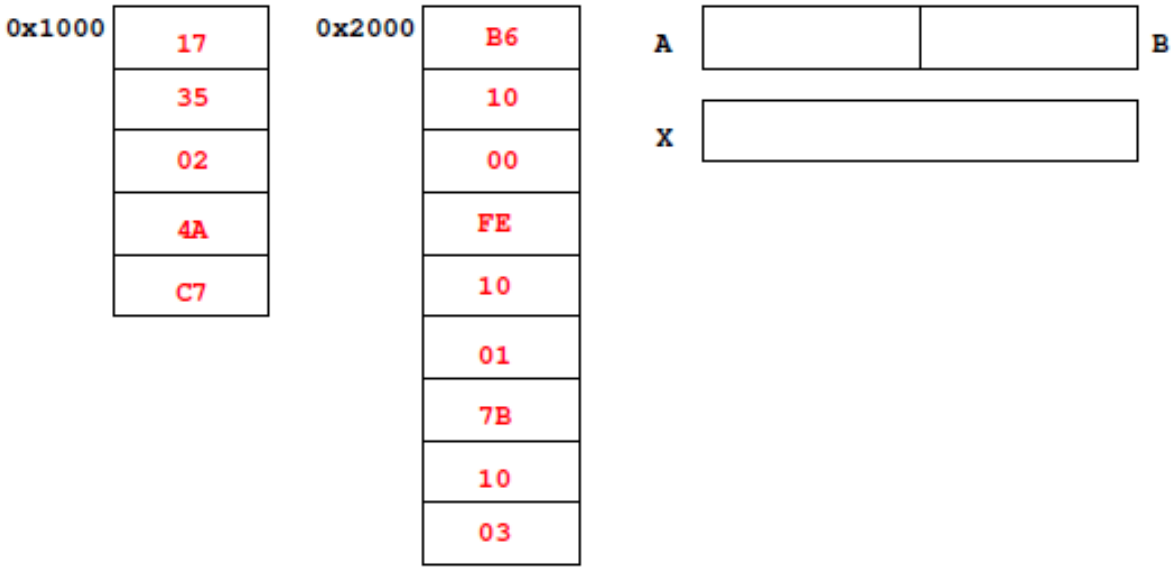
Instructions which give the 16−bit address to be accessed

LDAA $1000 ; ($1000) → A
B6 10 00 Effective Address: $1000

LDX $1001 ; ($1001:$1002) → X
FE 10 01 Effective Address: $1001

STAB $1003 ; (B) → $1003
7B 10 03 Effective Address: $1003

Effective address is specified by the two bytes following op code



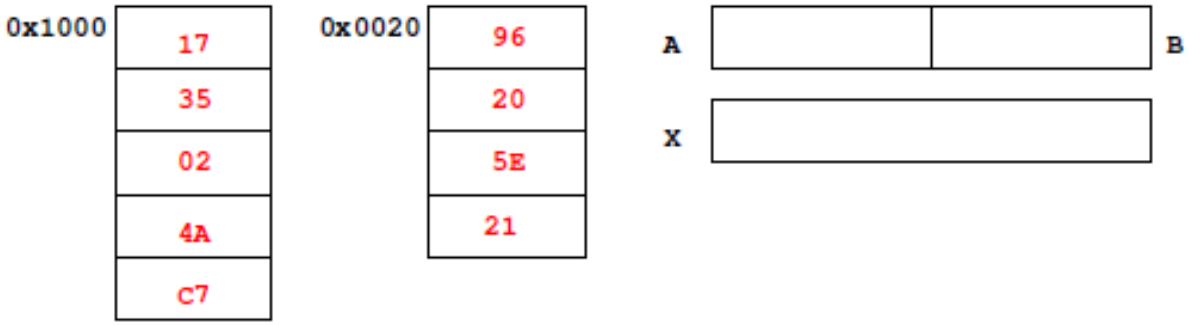
The Direct (DIR) addressing mode

Direct (DIR) Addressing Mode
Instructions which give 8 LSB of address (8 MSB all 0)

LDAA $20 ; ($0020) → A
96 20 Effective Address: $0020

STX $21 ; (X) → $0021:$0022
5E 21 Effective Address: $0021

8 LSB of effective address is specified by byte following op code



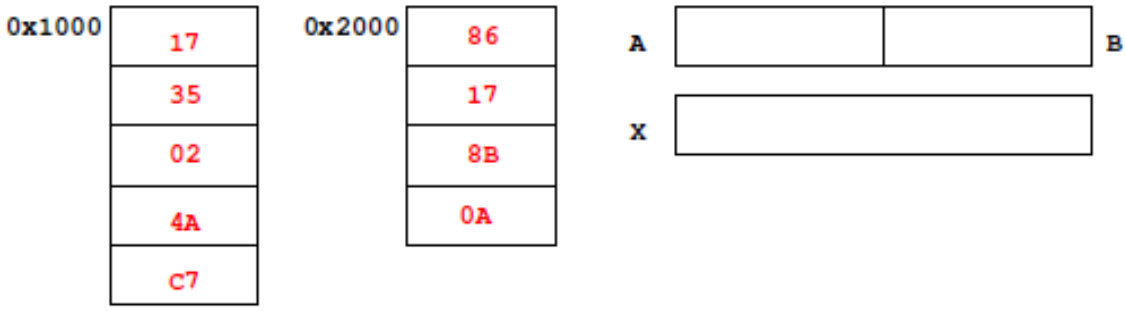
The Immediate (IMM) addressing mode

Value to be used is part of instruction

LDAA #$17 ; $17 → A
B6 17 Effective Address: PC + 1

ADDA #10 ; (A) + $0A → A
8B 0A Effective Address: PC + 1

Effective address is the address following the op code



The Indexed (IDX, IDX1, IDX2) addressing mode

Effective address is obtained from X or Y register (or SP or PC)
Simple Forms

```
LDAA 0,X        ; Use (X) as address to get value to put in A
A6 00           Effective address: contents of X

ADDA 5,Y        ; Use (Y) + 5 as address to get value to add
to
AB 45           Effective address: contents of Y + 5
```

More Complicated Forms

```
INC 2,X−        ; Post−decrement Indexed
                ; Increment the number at address (X),
                ; then subtract 2 from X
62 3E           Effective address: contents of X

INC 4,+X        ; Pre−increment Indexed
                ; Add 4 to X
                ; then increment the number at address (X)
62 23           Effective address: contents of X + 4
```

Table 3-1. M68HC12 Addressing Mode Summary

Addressing Mode	Source Format	Abbreviation	Description
Inherent	**INST** (no externally supplied operands)	INH	Operands (if any) are in CPU registers
Immediate	**INST** *#opr8i* or **INST** *#opr16i*	IMM	Operand is included in instruction stream 8- or 16-bit size implied by context
Direct	**INST** *opr8a*	DIR	Operand is the lower 8 bits of an address in the range $0000–$00FF
Extended	**INST** *opr16a*	EXT	Operand is a 16-bit address
Relative	**INST** *rel8* or **INST** *rel16*	REL	An 8-bit or 16-bit relative offset from the current pc is supplied in the instruction
Indexed (5-bit offset)	**INST** *oprx5,xysp*	IDX	5-bit signed constant offset from X, Y, SP, or PC
Indexed (pre-decrement)	**INST** *oprx3,–xys*	IDX	Auto pre-decrement x, y, or sp by 1 – 8
Indexed (pre-increment)	**INST** *oprx3,+xys*	IDX	Auto pre-increment x, y, or sp by 1 – 8
Indexed (post-decrement)	**INST** *oprx3,xys–*	IDX	Auto post-decrement x, y, or sp by 1 – 8
Indexed (post-increment)	**INST** *oprx3,xys+*	IDX	Auto post-increment x, y, or sp by 1 – 8
Indexed (accumulator offset)	**INST** *abd,xysp*	IDX	Indexed with 8-bit (A or B) or 16-bit (D) accumulator offset from X, Y, SP, or PC
Indexed (9-bit offset)	**INST** *oprx9,xysp*	IDX1	9-bit signed constant offset from X, Y, SP, or PC (lower 8 bits of offset in one extension byte)
Indexed (16-bit offset)	**INST** *oprx16,xysp*	IDX2	16-bit constant offset from X, Y, SP, or PC (16-bit offset in two extension bytes)
Indexed-Indirect (16-bit offset)	**INST** [*oprx16,xysp*]	[IDX2]	Pointer to operand is found at... 16-bit constant offset from X, Y, SP, or PC (16-bit offset in two extension bytes)
Indexed-Indirect (D accumulator offset)	**INST** [**D**,*xysp*]	[D,IDX]	Pointer to operand is found at... X, Y, SP, or PC plus the value in D

Different types of indexed addressing modes

(Note: We will not discuss indirect indexed mode)

INDEXED ADDRESSING MODES
(Does not include indirect modes)

	Example	Effective Address	Offset	Value in X After Done	Registers To Use
Constant Offset	LDAA n,X	(X)+n	0 to FFFF	(X)	X, Y, SP, PC
Constant Offset	LDAA -n,X	(X)-n	0 to FFFF	(X)	X, Y, SP, PC
Postincrement	LDAA n,X+	(X)	1 to 8	(X)+n	X, Y, SP
Preincrement	LDAA n,+X	(X)+n	1 to 8	(X)+n	X, Y, SP
Postdecrement	LDAA n,X-	(X)	1 to 8	(X)-n	X, Y, SP
Predecrement	LDAA n,-X	(X)-n	1 to 8	(X)-n	X, Y, SP

ACC Offset	LDAA A,X
	LDAA B,X
	LDAA D,X
		(X)+(A)
		(X)+(B)
		(X)+(D)
			0 to FF
			0 to FF
			0 to FFFF
				(X)	X, Y, SP, PC

The data books list three different types of indexed modes:

- Table 3.2 of the **S12CPUV2 Reference Manual** shows details

- **IDX:** One byte used to specify address
  - Called the postbyte
  - Tells which register to use
  - Tells whether to use autoincrement or autodecrement
  - Tells offset to use

- **IDX1:** Two bytes used to specify address
  - First byte called the postbyte
  - Second byte called the extension
  - Postbyte tells which register to use, and sign of offset
  - Extension tells size of offset

- **IDX2:** Three bytes used to specify address
  - First byte called the postbyte
  - Next two bytes called the extension
  - Postbyte tells which register to use
  - Extension tells size of offset

Table 3-2. Summary of Indexed Operations

Postbyte Code (xb)	Source Code Syntax	Comments rr; 00 = X, 01 = Y, 10 = SP, 11 = PC
rr0nnnnn	,r n,r –n,r	**5-bit constant offset** n = –16 to +15 r can specify X, Y, SP, or PC
111rr0zs	n,r –n,r	**Constant offset** (9- or 16-bit signed) z- 0 = 9-bit with sign in LSB of postbyte(s) –256 ≤ n ≤ 255 1 = 16-bit –32,768 ≤ n ≤ 65,535 if z = s = 1, 16-bit offset indexed-indirect (see below) r can specify X, Y, SP, or PC
111rr011	[n,r]	**16-bit offset indexed-indirect** rr can specify X, Y, SP, or PC –32,768 ≤ n ≤ 65,535
rr1pnnnn	n,–r n,+r n,r– n,r+	**Auto predecrement**, **preincrement**, **postdecrement**, or **postincrement**; p = pre-(0) or post-(1), n = –8 to –1, +1 to +8 r can specify X, Y, or SP (PC not a valid choice) +8 = 0111 … +1 = 0000 –1 = 1111 … –8 = 1000
111rr1aa	A,r B,r D,r	**Accumulator offset** (unsigned 8-bit or 16-bit) aa-00 = A 01 = B 10 = D (16-bit) 11 = see accumulator D offset indexed-indirect r can specify X, Y, SP, or PC
111rr111	[D,r]	**Accumulator D offset indexed-indirect** r can specify X, Y, SP, or PC

Indexed addressing mode instructions use a postbyte to specify index registers (X and Y), stack pointer (SP), or program counter (PC) as the base index register and to further classify the way the effective address is formed. A special group of instructions cause this calculated effective address to be loaded into an index register for further calculations:

- Load stack pointer with effective address (LEAS)
- Load X with effective address (LEAX)
- Load Y with effective address (LEAY)

Relative (REL) Addressing Mode

The relative addressing mode is used only in branch and long branch instructions.

Branch instruction: One byte following op code specifies how far to branch.

Treat the offset as a signed number; add the offset to the address following the current instruction to get the address of the instruction to branch to

(BRA) 20 35 PC + 2 + 0035 → PC

(BRA) 20 C7 PC + 2 + FFC7 → PC
PC + 2 − 0039 → PC

Long branch instruction: Two bytes following op code specifies how far to branch.

Treat the offset as an unsigned number; add the offset to the address following the current instruction to get the address of the instruction to branch to

(LBEQ) 18 27 02 1A If Z == 1 then PC + 4 + 021A → PC
If Z == 0 then PC + 4 → PC

When writing assembly language program, you don't have to calculate offset. You indicate what address you want to go to, and the assembler calculates the offset

Summary of MC9S12 addressing modes

ADDRESSING MODES

Name	Example	Op Code	Effective Address
INH Inherent	ABA	18 06	None
IMM Immediate	LDAA #$35	86 35	PC + 1
DIR Direct	LDAA $35	96 35	0x0035
EXT Extended	LDAA $2035	B6 20 35	0x2035

IDX Indexed
IDX1
IDX2
	LDAA 3,X
	LDAA 30,X
	LDAA 300,X
		A6 03
		A6 E0 13
		A6 E2 01 2C
			X + 3
			X + 30
			X + 300

IDX Indexed Postincrement	LDAA 3,X+	A6 32	X (X+3 -> X)
IDX Indexed Preincrement	LDAA 3,+X	A6 22	X+3 (X+3 -> X)
IDX Indexed Postdecrement	LDAA 3,X-	A6 3D	X (X-3 -> X)
IDX Indexed Predecrement	LDAA 3,-X	A6 2D	X-3 (X-3 -> X)
REL Relative	BRA $1050 LBRA $1F00	20 23 18 20 0E CF	PC + 2 + Offset PC + 4 + Offset

A few instructions have two effective addresses:

- **MOVB #$AA,$1C00** Move byte 0xAA (IMM) to address $1C00 (EXT)
- **MOVW 0,X,0,Y** Move word from address pointed to by X (IDX) to address pointed to by Y (IDX)

A few instructions have three effective addresses:

• **BRSET FOO,#$03,LABEL** Branch to LABEL (REL) if bits #$03 (IMM) of variable FOO (EXT) are set.

Using X and Y as Pointers

• Registers X and Y are often used to point to data.

• To initialize pointer use

```
ldx #table
```

not

```
ldx table
```

• For example, the following loads the address of table ($1000) into X; i.e., X will point to table:

```
ldx #table ; Address of table ⇒ X
```

The following puts the first two bytes of table ($0C7A) into X. X will not point to table:

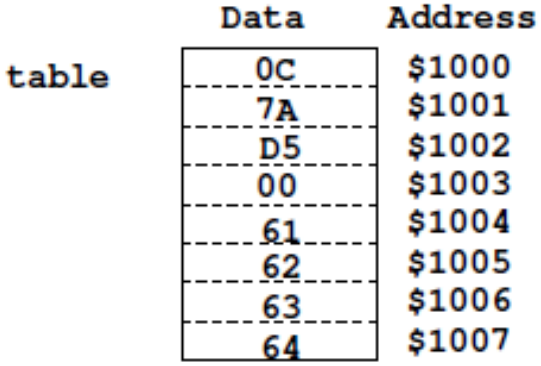
```
ldx table  ; First two bytes of table ⇒ X
```

• To step through table, need to increment pointer after use

```
ldaa 0,x
inx
```

or

```
ldaa 1,x+
```




```
        org   $1000
table:  dc.b  12,122,-43,0
        dc.b  'a'
        dc.b  'b'
        dc.b  'c'
        dc.b  'd'
```

Which branch instruction should you use?

Branch if A > B

Is 0xFF > 0x00?

If unsigned, 0xFF = 255 and 0x00 = 0,
so 0xFF > 0x00

If signed, 0xFF = −1 and 0x00 = 0,
so 0xFF < 0x00

Using unsigned numbers: **BHI**

Using signed numbers: **BGT**

Hand Assembling a Program

To hand-assemble a program, do the following:

1. Start with the org statement, which shows where the first byte of the program will go into memory.
(e.g., **org $2000** will put the first instruction at address **$2000**.)

2. Look at the first instruction. Determine the addressing mode used.
(e.g., **ldab #10** uses IMM mode.)

3. Look up the instruction in the **MC9S12 S12CPUV2 Reference Manual**, find the appropriate Addressing Mode, and the Object Code for that addressing mode. (e.g., **ldab IMM** has object code **C6 ii**.)

- **Table A.1 of S12CPUV2 Reference Manual** has a concise summary of the instructions, addressing modes, op-codes, and cycles.

4. Put in the object code for the instruction, and put in the appropriate operand. Be careful to convert decimal operands to hex operands if necessary. (e.g., **ldab #10** becomes **C6 0A**.)

5. Add the number of bytes of this instruction to the address of the instruction to determine the address of the next instruction.
(e.g., **$2000 + 2 = $2002** will be the starting address of the next instruction.)

Freescale HC12-Assembler
(c) Copyright Freescale 1987-2010

```
Abs. Rel.  Loc    Obj. code  Source line
---- ----  ------ ---------  -----------
   1    1
   2    2       0000 2000          ??
   3    3                              ??
   4    4  a002000 C60A                ??
   5    5  a002002 87                  ??
   6    6  a002003 0431 FC              ??
   7    7  a002006 3F                  ??
```

What is the corresponding assembly code?

Solution:

```
        org $2000
        ldab #10
loop:   clra
        dbne b,loop
        swi
```

MC9S12 Cycles

• 68HC12 works on **48 MHz clock**

• CPU cycles take 2 clock cycles, so clock is 24 MHz

• Each CPU cycle takes **41.7 ns** (1/24 MHz) to execute

• An instruction takes from **1** to **12** HCS12 cycles to execute

• You can determine how many cycles an instruction takes by looking up the CPU cycles for that instruction in the S12CPUV2 Core Users Guide.

– For example, **LDAA** using the **IMM** addressing mode shows one CPU cycle (of type P).

– **LDAA** using the **EXT** addressing mode shows three CPU cycles (of type **rPO**).

- Section 6.6 of the S12CPUV2 Reference Manual explains what the MC9S12 is doing during each of the different types of CPU cycles.

```
2000                  org $2000     ; Inst    Mode    Cycles
2000  C6 0A           ldab #10      ; LDAB    (IMM)    1
2002  87        loop: clra          ; CLRA    (INH)    1
2003  04 31 FC        dbne b,loop   ; DBNE    (REL)    3
2006  3F              swi           ; SWI              9
```

How many cycles does it take?
How long does it take to execute?

The program executes the **ldab #10** instruction **once** (which takes one cycle). It then goes through loop **10 times** (which has two instructions, one with one cycle and one with three cycles), and finishes with the swi instruction (which takes 9 cycles).

Total number of cycles:

$1 + 10 \times (1 + 3) + 9 = 50$

50 cycles = 50 × 41.7 ns/cycle = 2.08 μs

LDAB — Load B — LDAB

Operation

(M) ⇒ B
or
imm ⇒ B

Loads B with either the value in M or an immediate value.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise
Z: Set if result is $00; cleared otherwise
V: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LDAB #*opr8i*	IMM	C6 ii	P
LDAB *opr8a*	DIR	D6 dd	rPf
LDAB *opr16a*	EXT	F6 hh ll	rPO
LDAB *oprx0_xysppc*	IDX	E6 xb	rPf
LDAB *oprx9,xysppc*	IDX1	E6 xb ff	rPO
LDAB *oprx16,xysppc*	IDX2	E6 xb ee ff	frPP
LDAB [D,*xysppc*]	[D,IDX]	E6 xb	fIfrPf
LDAB [*oprx16,xysppc*]	[IDX2]	E6 xb ee ff	fIPrPf

CLRA — Clear A — CLRA

Operation: $0 \Rightarrow A$

Description: All bits in accumulator A are cleared to 0.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: 0; cleared
Z: 1; set
V: 0; cleared
C: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
CLRA	INH	87	O	O

DBNE Decrement and Branch if Not Equal to Zero DBNE

Operation: (Counter) – 1 ⇒ Counter
If (Counter) not = 0, then (PC) + $0003 + Rel ⇒ PC

Description: Subtract one from the specified counter register A, B, D, X, Y, or SP. If the counter register has not been decremented to zero, execute a branch to the specified relative destination. The DBNE instruction is encoded into three bytes of machine code including a 9-bit relative offset (–256 to +255 locations from the start of the next instruction).

IBNE and TBNE instructions are similar to DBNE except that the counter is incremented or tested rather than being decremented. Bits 7 and 6 of the instruction postbyte are used to determine which operation is to be performed.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code(1)	Access Detail HCS12	Access Detail M68HC12
DBNE *abdxys, rel9*	REL	`04 lb rr`	`PPP/PPO`	`PPP`

1. Encoding for `lb` is summarized in the following table. Bit 3 is not used (don't care), bit 5 selects branch on zero (DBEQ – 0) or not zero (DBNE – 1) versions, and bit 4 is the sign bit of the 9-bit relative offset. Bits 7 and 6 would be 0:0 for DBNE.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	DBNE A, *rel9*	`04 20 rr`	`04 30 rr`
B	001	DBNE B, *rel9*	`04 21 rr`	`04 31 rr`
D	100	DBNE D, *rel9*	`04 24 rr`	`04 34 rr`
X	101	DBNE X, *rel9*	`04 25 rr`	`04 35 rr`

SWI — Software Interrupt — SWI

Operation:

$(SP) - \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$
$1 \Rightarrow I$
$(SWI\ Vector) \Rightarrow PC$

Description: Causes an interrupt without an external interrupt service request. Uses the address of the next instruction after SWI as a return address. Stacks the return address, index registers Y and X, accumulators B and A, and the CCR, decrementing the SP before each item is stacked. The I mask bit is then set, the PC is loaded with the SWI vector, and instruction execution resumes at that location. SWI is not affected by the I mask bit. Refer to **Section 7. Exception Processing** for more information.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	1	–	–	–	–

I: 1; set

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
SWI	INH	3F	VSPSSPSsP(1)	VSPSSPSsP(1)

1. The CPU also uses the SWI processing sequence for hardware interrupts and unimplemented opcode traps. A variation of the sequence (VfPPP) is used for resets.