Chapter 4: Advanced Assembly Programming

The HCS12 Microcontroller

Han-Way Huang

Minnesota State University, Mankato

September 2009
Chapter Summery

- The same sequence of instructions often need to be executed in several places of the program.
- The subroutine mechanism enables the programmer to group the common instruction sequence into a subroutine.
- A subroutine can be called from many places.
- The processor saves the return address in the stack during a subroutine call.
- The subroutine restores the return address from the stack before it returns to the caller.
- The subroutine can receive parameters from the caller to perform operations and may return result to the caller.
- **Top-down design with hierarchical refinement** is the most effective and popular software development methodology.
**Introduction**

- Program = data structures + algorithm
- Data structures to be discussed
  1. stacks: a **last-in-first-out** data structure
  2. arrays: a set of elements of the same type
  3. strings: a sequence of characters terminated by a special character

**Stack**

![Diagram of the HCS12 stack](image)

Figure 4.1 Diagram of the HCS12 stack
HCS12 Support for the Stack Data Structure

- A 16-bit stack pointer (SP)
- Instructions and addressing mode

Table 4.1 HCS12 push and pull instructions and their equivalent load and store instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Equivalent instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>psha</td>
<td>push A into the stack</td>
<td>staa 1, -SP</td>
</tr>
<tr>
<td>pshb</td>
<td>push B into the stack</td>
<td>stab 1, -SP</td>
</tr>
<tr>
<td>pshc</td>
<td>push CCR into the stack</td>
<td>none</td>
</tr>
<tr>
<td>pshd</td>
<td>push D into stack</td>
<td>std 2, -SP</td>
</tr>
<tr>
<td>pshx</td>
<td>push X into the stack</td>
<td>stx 2, -SP</td>
</tr>
<tr>
<td>pshy</td>
<td>push Y into the stack</td>
<td>sty 2, -SP</td>
</tr>
<tr>
<td>pula</td>
<td>pull A from the stack</td>
<td>ld醛a 1, SP+</td>
</tr>
<tr>
<td>pulb</td>
<td>pull B from the stack</td>
<td>ldab 1, SP+</td>
</tr>
<tr>
<td>pulc</td>
<td>pull CCR from the stack</td>
<td>none</td>
</tr>
<tr>
<td>puld</td>
<td>pull D from the stack</td>
<td>ldd 2, SP+</td>
</tr>
<tr>
<td>pulx</td>
<td>pull X from the stack</td>
<td>ldx 2, SP+</td>
</tr>
<tr>
<td>puly</td>
<td>pull Y from the stack</td>
<td>ldy 2, SP+</td>
</tr>
</tbody>
</table>
Example 4.1 Derive the stack contents after the execution of the following instruction sequence:

```
lds #$15C0
ldaa #$20
psha
ldab #40
pshb
ldx #0
pshx
```

Solution:

The contents of the HCS 12 stack after the execution of the above sequence are shown in Figure 4.2.
What is a Subroutine?

- A sequence of instructions that can be called from many places of the program
- The program flow during a subroutine call is shown in Figure 4.3.
- Saving and restoring return address is the key for the subroutine mechanism to work.

![Diagram of subroutine call and return](image)

Figure 4.3 Program flow during a subroutine call
Instructions that Support Subroutine Call

- **bsr <sub>:** branch subroutine
  <sub> is specified in relative addressing mode and is in the range of -128 to +127 bytes from the instruction immediately after the **bsr <sub>** instruction. The return address is pushed into the stack.

- **jsr <sub>:** jump subroutine
  <sub> is specified in direct, extended, indexed, and indexed indirect mode. The subroutine being called can be within the range of 64 kB. The return address is pushed into the stack.

- **call <sub>:** call subroutine
  The <sub> field specifies the page number and the address of the subroutine within the page to be called. The page number is to be loaded into the PPAGE register.

- **rts:** return from subroutine
  Retrieve the return address from the stack and return to the caller.

- **rtc:** return from call
  This instruction retrieve the page number and the return address from the stack.
Example 4.2 Write a subroutine that converts a 16-bit binary number into a BCD ASCII string. The 16-bit number is passed in D and the pointer to the buffer to hold the string is passed in Y.

Solution:
Let num, ptr, quo, and rem represent the number to be converted pointer, quotient, and remainder. The procedure is as follows:

Step 1
Push a 0 into the stack. quo ← num.

Step 2
rem ← quo % 10; quo ← quo / 10.

Step 3
Push the sum of $30 and rem into the stack.

Step 4
If (quo == 0) goto Step 5; else goto step 2.

Step 5
Pull a byte from stack and save it in the location pointed to by ptr.

Step 6
ptr ← ptr + 1.

Step 7
If the byte pulled in Step 5 is NULL, stop; else goto Step 5.
bin2dec
  pshx ; save X in stack
  movb #0,1,-SP ; push NULL character into stack
divloop
  ldx #10 ; divide the number by 10
  idiv ;
  addb #$30 ; convert the remainder to ASCII code
  pshb ; push it into the stack
  xgdx ; swap quotient to D
  cpd #0 ; if quotient is 0, then prepare to pull out
  beq revloop ; the decimal digit characters
  bra divloop ; quotient is not 0, continue to perform divide-by-10
revloop
  pula ; start to reverse the string from stack
  staa 1,y+ ; save ASCII string in the buffer
  cmpa #0 ; reach the NULL pushed previously?
  beq done ; if yes, then done
  bra revloop ; continue to pull
done
  pulx ; restore the index register X
  rts
Issues related to Subroutine Calls

1. Parameter Passing
   - Use CPU registers
   - Use stack
   - Use global memory

2. Result Returning
   - Use CPU registers
   - Use stack
   - Use global memory

3. Local Variable Allocation
   - Allocated in the stack
   - Use the instruction `leas -n,SP` to allocate `n bytes` of space in stack
   - Use the instruction `leas n, SP` to deallocate `n bytes` from stack
Saving CPU Registers in Stack

- The subroutine may use some CPU registers that were used by its caller.
- The subroutine must save these registers in order to make sure the caller program can obtain correct result.
- The subroutine must restore the saved registers before returning to the caller.
- The order of register restoration must be reversed to that to which registers are saved.

if the saving order is

```
pshx
pshy
pshy
pshd
```

then the restoring order is

```
puld
puly
pulx
```
Stack Frame

- The region in the stack that holds incoming parameters, the subroutine return address, local variables, and saved registers is referred to as stack frame.
- The stack frame is also called **activation record**.

Figure 4.9 Structure of the 68HC12 stack frame
Example 4.3 Draw the stack frame for the following program segment after the `leas -10,sp` instruction is executed:

```
ldd #$1234
pshd
ldx #$4000
pshx
jsr sub_xyz
...
sub_xyz pshd
pshx
pshy
leas -10,sp
...
```

**Solution:** The stack frame is shown in Figure 4.10.
Subroutines with Local Variables

Example 4.4 Write a subroutine that can convert a BCD ASCII string to a binary number and leave the result in double accumulator D. The ASCII string represents a number in the range of $-2^{15} \sim 2^{15} - 1$. A pointer to the string is passed to this subroutine in X.

Solution:
The subroutine will return an error indication using the CY flag of the CCR register. Let $\text{in\_ptr}$, $\text{sign}$, $\text{error}$, and $\text{number}$ represent the pointer to the BCD string, sign flag, error flag, and the number represented by the BCD string.

The algorithm is:
Step 1
$\text{sign} \leftarrow 0$; $\text{error} \leftarrow 0$; $\text{number} \leftarrow 0$.

Step 2
If the character pointed to by $\text{in\_ptr}$ is the NULL character, then go to step 4.
else if the character is not a BCD digit, then:
\hspace{1em} $\text{error} \leftarrow 1$;
\hspace{1em} go to step 4;
else
\hspace{1em} $\text{number} \leftarrow \text{number} \times 10 + \text{m[\text{in\_ptr}] - $30}$;
\hspace{1em} $\text{in\_ptr} \leftarrow \text{in\_ptr} + 1$;
\hspace{1em} go to step 3;
**Step 4**

If sign = 1 and error = 0, then

number \( \rightarrow \) two’s complement of number;

else

stop;

This subroutine allows local variables in the stack and its stack is shown in Figure 4.6.

![Stack frame of Example 4.4](image)
```assembly
c; Include the HCS12/MC9S12 Microcontroller header file
#include "c:\miniide\hcs12.inc"

; Constants
minus     equ $2D ; ASCII code of minus sign
dummy     equ 0 ; offset of dummy to hold a 0
pdVal     equ 1 ; offset of present digit value from SP in stack
val       equ 2 ; offset of the 2-byte binary value of the BCD string
            ; from SP in stack
sign      equ 4 ; offset of the sign from SP in stack
err       equ 5 ; offset of error flag from SP in stack
locVar    equ 6
            org $1000
strBuf    dc.b "-9889",0 ; input ASCII to be converted
result    ds.w 1
            org $1500
start     lds #$1500
            ldx #strBuf
            jsr bcd2bin
            std result
            swi

; This subroutine converts a BCD string into its equivalent binary value and also uses the
; CY flag to indicate error condition. The CY flag is set to 1 to indicate error.
```

H. Huang Transparency No.4-16

Copyright © 2010 Delmar Cengage Learning
bcd2bin

leas -locVar,SP ; allocate 4 bytes for local variables
movw #0,val,SP ; initialize accumulated value to 0
movb #0,dummy,SP
movb #0,sign,SP ; initialize sign to positive
movb #0,err,SP ; clear error flag initially
ldaa 0,x ; check the first character
cmpa #minus ; is the first character a minus sign?
bne GEZero ; branch if not minus
movb #1,sign,SP ; set the sign to 1
inx ; move the pointer

GEZero

ldab 1,x+ ; is the current character a NULL character?
lbeq done ; yes, we reach the end of the string
cmpb #$30 ; is the character not between 0 to 9?
blo inErr ; "
cmpb #$39 ; "
bhi inErr ; "
subb #$30 ; convert to the BCD digit value
stab pdVal,SP ; save the current digit value
ldd val,SP ; get the accumulated value
ldy #10
emul ; perform 16-bit by 16-bit multiplication
add
dummy,SP ; add the current digit value
std
val,SP ; save the sum
bra
GEZero

inErr
movb
#1,err,SP ; set the error flag to indicate error
bra
chkout
done
ldaa
sign,SP ; check to see if the original number is negative
beq
chkout
ldd
#$FFFF ; convert to two's complement format
subd
val,SP ; if the number is negative
addd
#1 ; "
std
val,SP ; "

chkout
ldaa
err,SP ; check the error flag
beq
clrErr ; go to clear CY flag before return
sec ; clear the C flag
bra
dealloc
clrErr
clc ; set the C flag
dealloc
ldd
val,SP
leas
locVar,SP ; deallocate local variables
puly ; restore Y
rts
;
org
$FFFF ; uncomment these two lines for CodeWarrior
;
dc.w
start
end
Bubble Sort
- Sorting is useful for improving the searching speed when an array or a file needs to be searched many times.
- Bubble sort is a simple but inefficient sorting method.

Example 4.13 Write a subroutine to implement the bubble sort algorithm and a sequence of instructions for testing the subroutine.

Solution: The algorithm of bubble sort is shown in Figure 4.7. The bubble sort subroutine uses the following local variables:

- **Buf**: a buffer for swapping elements
- **InOrder**: a flag indicating whether the array is in order
- **Inner**: number of comparisons remained to be performed in the current iteration
- **Iteration**: number of iterations remained to be performed
Figure 4.7 Logic flow of bubble sort
Stack Frame for bubble Sort

Figure 4.8 Stack frame for bubble sort
arr equ 13 ; distance of the variable arrayX from stack top
arcnt equ 12 ; distance of the variable arcnt from stack top
buf equ 3 ; distance of local variable buf from stack top
inOrder equ 2 ; distance of local variable inOrder from stack top
inner equ 1 ; distance of local variable inner from stack top
iteration equ 0 ; distance of local variable iteration from stack top
true equ 1
false equ 0
n equ 30 ; array count
local equ 4 ; number of bytes used by local variables
org $1000 ; test data
arrayX dc.b 3,29,10,98,54,9,100,104,200,92,87,48,27,22,71
org $1500
start lds #$1500 ; initialize stack pointer
ldx #arrayX
pshx
ldaa #n
psha
jsr bubble
leas 3,sp ; deallocate space used by outgoing parameters
; bra $ ; uncomment this instruction under CodeWarrior
swi ; break to D-Bug12 monitor
bubble    pshd
pshy
pshx
leas -local,sp ; allocate space for local variables
ldaa arcnt,sp ; compute the number of iterations to be performed
dec a ;
staa iteration,sp; 

ploop    ldaa #true ; set array inOrder flag to true before any iteration
staa inOrder,sp ;
ldx arr,sp ; use index register X as the array pointer
ldaa iteration,sp; initialize inner loop count for each iteration
staa inner,sp ;

cloop    ldaa 0,x ; compare two adjacent elements
cmpa 1,x ;
bls looptst
; the following five instructions swap the two adjacent elements
staa buf,sp ; swap two adjacent elements
ldaa 1,x ;
staa 0,x ;
ldaa buf,sp ;
staa 1,x ;
ldaa #false ; reset the inOrder flag
staa inOrder,sp ;
looptst inx
  dec inner,sp
  bne cloop
  tst inOrder,sp ; test array inOrder flag after each iteration
  bne done
  dec iteration,sp
  bne ploop

; the following instruction deallocates local variables
done leas local,sp ; de-allocate local variables
  pulx
  puly
  puld
  rts

; org $FFFE ; uncomment this line for CodeWarrior
; dc.w start ; uncomment this line for CodeWarrior
end
Binary Search Subroutine

- Divide the array into three parts: upper half, middle element, and lower half.
- Let $\text{min}$, $\text{max}$, $\text{mean}$ be the minimum, maximum, and middle indices of the subarray to be searched.
- The algorithm of binary search is as follows:

**Step 1**

$\text{min} \leftarrow 0$; $\text{max} = n - 1$;

**Step 2**

If $(\text{max} < \text{min})$, stop.

**Step 3**

$\text{Mean} \leftarrow (\text{min} + \text{max})/2$

**Step 4**

If ($\text{key} == \text{arr}[\text{mean}]$), then key is found and exit.

**Step 5**

If ($\text{key} < \text{arr}[\text{mean}]$), then set $\text{max}$ to $\text{mean} - 1$ and go to Step 2.

**Step 6**

if ($\text{key} < \text{arr}[\text{mean}]$), then set $\text{min}$ to $\text{mean} + 1$ and go to Step 2.
Figure 4.9 Stack frame for binary search

```assembly
n equ 30  ; array count
srch equ 69  ; key to be searched
mean equ 0  ; stack offset for local variable mean
min equ 1  ; stack offset for local variable min
max equ 2  ; stack offset for local variable max
key equ 8  ; stack offset for local variable key
arrCnt equ 9  ; stack offset for local variable arrCnt
arrBas equ 10  ; stack offset for local variable arrBas
locvar equ 3  ; number of bytes for local variables
org $1000
result ds.b 1  ; search result
```
org   $1500
lds   #$1500
movw  #arr,2,-SP ; pass array base address
movb  #n,1,-SP  ; pass array count
movb  #srch,1,-SP ; pass key for search
jsr   binsearch
leas  4,SP       ; de-allocate space used in passing parameters
staa  result
;    bra   $       ; uncomment this instruction for CodeWarrior
swi

binSearch    pshx   ; save X in the stack
             ; save B in the stack
             ; allocate space for locVar variables
movb  #0,min,SP ; initialize min to 0
ldaa  arrCnt,SP ; initialize max to arCnt - 1
deca  ;
staa  max,SP  ;
ldx   arrBas,SP ; use X as the pointer to the array
loop  ldab  min,SP  ; is search over yet?
cmpb  max,SP  ;
lbhi  notfound  ; if min > max, then not found (unsigned comparison)
The HCS12/MC9S12 Microcontroller

```
addb max,SP ; compute mean
lsrb ; "
stab mean,SP ; save mean
ldaa b,x ; get a copy of the element arr[mean]
cmpa key,SP ; compare key with array[mean]
beq found ; found it?
bhi searchLO ; continue to search in lower half
ldaa mean,SP ; prepare to search in upper half
inca ; "
staa min,SP ; "
bra loop
searchLO ldaa mean,SP ; set up indices range for searching in the
deca ; lower half
staa max,SP ; "
bra loop
found ldaa #1
bra exit
notfound ldaa #0
exit leas locVar,SP
pulb pulx rts
```
<table>
<thead>
<tr>
<th>arr</th>
<th>db</th>
<th>1,3,6,9,11,20,30,45,48,60</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>db</td>
<td>61,63,64,65,67,69,72,74,76,79</td>
</tr>
<tr>
<td></td>
<td>db</td>
<td>80,83,85,88,90,110,113,114,120,123</td>
</tr>
<tr>
<td>;</td>
<td>org</td>
<td>$FFFFFF</td>
</tr>
<tr>
<td>;</td>
<td>dc.w</td>
<td>start</td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Subroutine that perform 32-bit division

![Diagram of 32-bit division](image)

Figure 4.10 Conceptual hardware for implementing the repeated subtraction method
Algorithm for Multi-byte Division

Step 1
icnt ← n; R ← 0; Q ← dividend; P ← divisor.

Step 2
Shift the register pair (R, Q) one bit to the left.

Step 3
Subtract P from R, put the result back to R if the result is nonnegative.

Step 4
If the result of Step 2 is negative, then set the lsb of Q to 0. Otherwise, set the lsb to 1.

Step 5
icnt ← icnt – 1.

Step 6
If (icnt == 0) then stop; else go to Step 2

Example 4.7 Write a subroutine to implement the division algorithm using the repeated subtraction method for a 32-bit unsigned dividend and divisor. The caller of the subroutine allocate space in the stack for this subroutine to return the quotient and remainder and then push the dividend and divisor into the stack before calling this subroutine.
The HCS12/MC9S12 Microcontroller

Figure 4.11 Stack frame for Example 4.7

buf equ 0 ; distance of buf from the top of the stack
i equ 4 ; distance of i from the top of the stack
R equ 5 ; distance of R from the top of the stack
Q equ 9 ; distance of Q from the top of the stack
divisor equ 21 ; distance of divisor from the top of the stack
dividend equ 25 ; distance of dividend from the top of the stack
quo equ 29 ; distance of quo from the top of the stack
rem equ 33 ; distance of rem from the top of the stack
locVar equ 13 ; number of bytes for local variables
dvdendHI  equ  $42    ; dividend to be tested
dvdendLO  equ  $4c15  ;     

dvsorHI   equ  $0     ; divisor to be tested
dvsorLO   equ  $64    ;     

org  $1000

quotient ds.b  4     ; memory locations to hold the quo
remain ds.b  4     ; memory locations to hold the remainder

org  $1500 ; starting address of the program

start lds  #$1500  ; initialize stack pointer
leas  -8,SP ; make a hole of 8 bytes to hold the result

ldd  #dvdendLO
pshd

ldd  #dvdendHI
pshd

ldd  #dvsorLO
pshd

ldd  #dvsorHI
pshd

jsr  div32 ; call the divide subroutine

leas  8,SP ; de-allocate space used by dividend & divisor

movw  2,sp+,quotient ; pull upper half of quotient

movw  2,sp+,quotient+2 ; pull lower half of quotient
movw 2,sp+, quotient ; pull upper half of the quotient from the stack
movw 2,sp+, quotient+2 ; pull lower half of the quotient from the stack
bra 2

div32 pshd
pshx
pshy
leas -locVar,sp ; allocate space for local variables
ldd #0
std R, sp ; initialize R to 0
std R+2,sp ; 
ldd dividend,sp ; place dividend in register Q
std Q, sp
ldd dividend+2,sp
std Q+2,sp
movb #32, i, sp ; initialize loop count
dloop lsl Q+3, sp ; shift (R,Q) to the left by 1 bit
rol Q+2, sp ; 
rol Q+1, sp ; 
rol Q, sp ; 
rol R+3, sp ; 
rol R+2, sp ; 
rol R+1, sp ; 
rol R, sp ;
ldd    R+2,sp
subd   divisor+2,sp
std    buf+2,sp
ldaa   R+1,sp
sbca   divisor+1,sp
staa   buf+1,sp
ldaa   R,sp
sbca   divisor,sp
bc           smaller

; the following six instructions store the difference back to R register
staa   R,sp
ldaa   buf+1,sp
staa   R+1,sp
ldd    buf+2,sp
std    R+2,sp
bset   Q+3,sp,$01 ; set the least significant bit of Q register to 1
bra     looptest
smaller bclr  Q+3,sp,$01 ; set the least significant bit of Q register to 0
looptest dec   i,sp
lbne    loop

; the following four instructions copy the remainder into the hole in the stack
lddd   R,sp
std    rem,sp
ldd R+2,sp
std rem+2,sp

; the following four instructions copy the quotient into the hole in the stack
  ldd Q,sp
  std quo,sp
  ldd Q+2,sp
  std quo+2,sp
leas locVar,sp ; deallocate local variables
ply
pulx
puld
rts

;  org $FFFE ; uncomment this line for CodeWarrior
;  dc.w start ; uncomment this line for CodeWarrior
end
Finding the Square Root

The successive approximation method can be used to find the square root of an integer.

**Step 1**
sar ← 0; mask ← $8000$; lpcnt ← 16; temp ← 0.

**Step 2**
temp ← sar OR mask

**Step 3**
If ((temp * temp) ≤ num)
    sar ← temp;

**Step 4**
mask ← mask >> 1 (shift right one bit)

**Step 5**
lpcnt ← lpcnt – 1

**Step 6**
If (lpcnt == 0) then stop; else go to Step 2.

**Example 4.8** Write a subroutine that implements the successive approximation method to find the square root of a 32-bit integer.
#include "c:\miniide\hcs12.inc"

mask equ 0 ; stack offset of the variable mask from SP
sar equ 2 ; stack offset of the variable sar from SP
temp equ 4 ; stack offset of the variable temp from SP
lpcnt equ 6 ; stack offset of one-byte loop count from SP
q_hi equ 13 ; stack offset of the upper and lower halves of the
q_lo equ 15 ; number Q we want to find the square root from SP
locVar equ 7
testHi equ $00 ; upper half of the test number (q)
testLo equ $7BC4 ; lower half of the test number (q)
org $1000
sqroot ds.w 1 ; square root
org $1500

start lds #$1500
movw #testLo,2,-SP ; push testHi into stack
movw #testHi,2,-SP ; push testLo into stack
jsr findsqr
std sqroot ; save the returned square root
leas 4,SP ; de-allocate the space used in passing parameters
; bra $ ; uncomment this line for CodeWarrior
swi

; ------------------------------------------------------------------------------------------------------

findsqr pshx
pshy
leas -locVar,SP ; allocate space for local variables
movw #0,sar,SP ; initialize SAR to 0
movw #$8000,mask,SP ; initialize mask to $8000
movb #16,lpcnt,SP ; initialize loop count to 16

iloop ldd mask,SP ; get the mask
oraa sar,SP ; set a bit in SAR to 1
orab sar+1,SP ;

std temp,SP ; save a copy of your guess
tfr D,Y ; compute sar * sar
emul ;

H. Huang Transparency No.4-38
The square root computed using this method **tends to be smaller** than the actual square root and hence may not be the closest approximation and the user should compare \((\text{sar} + 1)\) and \(\text{sar}\) to find the closest approximation.
Subroutine for Prime Testing

- The most efficient method for testing whether an integer is a prime is to divide the given number by all the prime numbers from 2 to the square root of the given number.
- Since the prime numbers from 2 to the square root of the given number are not available, we can test divide the given number by all the numbers from 2 to its square root.

Example 4.9 Write a subroutine that can test whether an unsigned integer is a prime number.

Solution:

```assembly
test_hi equ $0638 ; number to be tested for prime
test_lo equ $F227 ;
org $1000
isprime ds.b 1
org $1500
start lds #$1500 ; set up stack pointer
movw #test_lo,2,-SP ; push the lower half of the test number
movw #test_hi,2,-SP ; push the upper half of the test number
jsr PrimeTest
staa isprime
; bra $ ; uncomment this line for CodeWarrior
swi
```
### Prime Number Test

```assembly
ii    equ    0    ; stack offset from SP of loop index
tlimit  equ  2    ; stack offset from SP of test limit
pNumHi  equ  10   ; stack offset from SP of upper half of test number
pNumLo  equ  12   ; stack offset from SP of lower half of test number
pLocal  equ  4    ; number of bytes used by local variables

primeTest pshx
  pshy
  leas    -pLocal,SP  ; allocate space for local variables
  ldaa    pNumLo+1,SP ; check if the number is even (if bit 0 is 0)
  anda    #$01        ;
  beq    nonPRI       ;
  lea     -pLocal,SP  ;
  ldaa    pNumHi,SP   ;
  pshd    pNumHi,SP   ;
  cpd     #0          ;
  bne    testPR       ; upper half nonzero, then enter normal test
  ldd    pNumLo,SP    ; if upper half is 0, then test lower half
  cpd     #0          ; is lower half equal to 0?
  beq    nonPri       ; 0 is not a prime
  cpd     #1          ; is lower half equal to 1
  beq    nonPri       ; 1 is not a prime

  testPR  ldd    pNumLo,SP  ; find the square root of Num
  ldx    pNumHi,SP    ;
  pshd    ;
  pshx    ;
```
The HCS12/MC9S12 Microcontroller

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jsr findsqr</td>
<td>; “</td>
</tr>
<tr>
<td>leas 4,SP</td>
<td>; de-allocate space for passing parameters</td>
</tr>
<tr>
<td>std tlimit,SP</td>
<td>; save returned value as the prime test limit</td>
</tr>
<tr>
<td>movw #2,ii,SP</td>
<td>; initialize test divide number to 3</td>
</tr>
<tr>
<td>divLoop ldd ii,SP</td>
<td>;</td>
</tr>
<tr>
<td>cpd tlimit,SP</td>
<td>; has test divided all numbers up to tlimit?</td>
</tr>
<tr>
<td>bhi isPRI</td>
<td>; the number is prime</td>
</tr>
<tr>
<td>ldd pNumLo,SP; divide Num by ii</td>
<td>;</td>
</tr>
<tr>
<td>ldx pNumHi,SP; “</td>
<td>;</td>
</tr>
<tr>
<td>ldy ii,SP</td>
<td>; “</td>
</tr>
<tr>
<td>leas -8,SP</td>
<td>; “</td>
</tr>
<tr>
<td>pshd</td>
<td>; “ (push pNumLo)</td>
</tr>
<tr>
<td>pshx</td>
<td>; “ (push pNumHi)</td>
</tr>
<tr>
<td>pshy</td>
<td>; “ (push ii)</td>
</tr>
<tr>
<td>movw #0,2,-SP</td>
<td>; “ (push 0 to the stack)</td>
</tr>
<tr>
<td>jsr div32</td>
<td>; “ (call the divide subroutine)</td>
</tr>
<tr>
<td>leas 14,SP</td>
<td>; de-allocate the space used by outgoing parameters</td>
</tr>
<tr>
<td>puld</td>
<td>; get the lower two bytes of the remainder</td>
</tr>
<tr>
<td>cpd #0</td>
<td>; is remainder equal to 0?</td>
</tr>
<tr>
<td>beq nonPRI</td>
<td>; If remainder equals 0, then Num is not a prime</td>
</tr>
<tr>
<td>ldd ii,SP</td>
<td>; test divide the next higher integer</td>
</tr>
<tr>
<td>addd #1</td>
<td>; “</td>
</tr>
<tr>
<td>std ii,SP</td>
<td>; “</td>
</tr>
</tbody>
</table>
The HCS12/MC9S12 Microcontroller

bra divLoop

isPRI ldaa #1
bra exitPT

nonPRI ldaa #0

exitPT leas pLocal,SP
pulypulxrts

#include "c:\miniide\findsqr.asm"
#include "c:\miniide\div32.asm"

; org $FFFE ; uncomment this line for CodeWarrior
; dc.w start ; uncomment this line for CodeWarrior
end
Using the D-Bug12 Functions to Perform I/O Operations

Table 4.2 D-Bug12 monitor (version 4.x.x) routines

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Function</th>
<th>Pointer address</th>
</tr>
</thead>
<tbody>
<tr>
<td>far main ()</td>
<td>Start of D-Bug12</td>
<td>$EE80</td>
</tr>
<tr>
<td>getchar ()</td>
<td>Get a character from SCI 0 or SCI1</td>
<td>$EE84</td>
</tr>
<tr>
<td>putchar ()</td>
<td>Send a character out to SCI 0 or SCI1</td>
<td>$EE86</td>
</tr>
<tr>
<td>printf ()</td>
<td>Formatted string output - translates binary values to string</td>
<td>$EE88</td>
</tr>
<tr>
<td>far GetCmdLine ()</td>
<td>Get a line of input from the user</td>
<td>$EE8A</td>
</tr>
<tr>
<td>far sscanhex ()</td>
<td>Convert ASCII hex string to a binary integer</td>
<td>$EE8E</td>
</tr>
<tr>
<td>isxdigit ()</td>
<td>Check if a character (in B) is a hex digit</td>
<td>$EE92</td>
</tr>
<tr>
<td>toupper ()</td>
<td>Convert lowercase characters to uppercase</td>
<td>$EE94</td>
</tr>
<tr>
<td>isalpha ()</td>
<td>Check if a character is alphabetic</td>
<td>$EE96</td>
</tr>
<tr>
<td>strlen ()</td>
<td>Returns the length of a NULL-terminated string</td>
<td>$EE98</td>
</tr>
<tr>
<td>strcpy ()</td>
<td>Copy a NULL-terminated string</td>
<td>$EE9A</td>
</tr>
<tr>
<td>far out2hex ()</td>
<td>Output 8-bit number as two ASCII hex characters</td>
<td>$EE9C</td>
</tr>
<tr>
<td>far out4hex ()</td>
<td>Output a 16-bit number as four ASCII hex characters</td>
<td>$EEA0</td>
</tr>
<tr>
<td>SetUserVector ()</td>
<td>Set up a vector to a user's interrupt service routine</td>
<td>$EEA4</td>
</tr>
<tr>
<td>far WriteEEByte ()</td>
<td>Write a byte to the on-chip EEPROM memory</td>
<td>$EEA6</td>
</tr>
<tr>
<td>far EraseEE ()</td>
<td>Bulk erase the on-chip EEPROM memory</td>
<td>$EEAA</td>
</tr>
<tr>
<td>far ReadMem ()</td>
<td>Read data from the HCS 12 memory map</td>
<td>$EEAE</td>
</tr>
<tr>
<td>far WriteMem ()</td>
<td>Write data to the HCS 12 memory map</td>
<td>$EEB2</td>
</tr>
</tbody>
</table>
- All functions listed in Table 4.2 are written in C language.

- The first parameter to the function is passed in accumulator D. The remaining parameters are pushed onto the stack in the reverse order they are listed in the function declaration.

- Parameters of type char will occupy the lower order byte of a word pushed onto the stack.

- Parameters pushed onto the stack before the function is called remain on the stack when the function returns. It is the responsibility of the caller to remove passed parameters from the stack.

- All 8- and 16-bit values are returned in accumulator D. A returned value of type char is returned in accumulator B.

**int getchar (void)**

Pointer address: $EE84

Returned value: 8-bit character in accumulator B
Adding the following instruction sequence will read a character from SCI0 port:

```assembly
getchar equ $EE84

... jsr [getchar,PCR]
```

**int putchar(int)**

- Pointer address: $EE86
- Incoming parameter: character to be output in accumulator B
- Returned value: the character that was sent (in B)

- This function outputs a character to serial communication port SCI0.
- The character to be output should be placed in accumulator B.
- The following instruction sequence will output the character A to serial port SCI0 when the program is running on the **Dragon12** demo board.

```assembly
putchar equ $EE86

... ldab #$41 ; place the ASCII code of A in accumulator B
jsr [putchar,PCR]
...```
int printf(char *format,...)
Pointer address: $EE88
Incoming parameters: zero or more integer data to be output on the stack, D contains the
address of the format string. The format string must be terminated with a zero.
Returned value: number of characters printed in D.

- This function is used to convert, format, and print its argument as standard output under
  the control of the format string pointed to by \textit{format}.
- All except floating-point data types are supported.
- The format string contains two basic types of objects:

1. ASCII characters which will be copied directly to the display device.
2. Conversion specifications. Each conversion specification begins with a percent sign (%).
3. Optional formatting characters may appear between the percent sign and ends with a
   single conversion character in the following order:

\[-\][<FieldWidth>][.][<Precision>][h | l]
The Meaning of Optional Characters

Table 4.3 Optional formatting characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-(minus sign)</td>
<td>Left justifies the converted argument.</td>
</tr>
<tr>
<td>FieldWidth</td>
<td>Integer number that specifies the minimum field width for the converted argument. The argument will be displayed in a field at least this wide. The displayed argument will be padded on the left or right if necessary.</td>
</tr>
<tr>
<td>. (period)</td>
<td>Separates the field width from the precision.</td>
</tr>
<tr>
<td>Precision</td>
<td>Integer number that specifies the maximum number of characters to display from a string or the minimum number of digits for an integer.</td>
</tr>
<tr>
<td>h</td>
<td>To have an integer displayed as a short.</td>
</tr>
<tr>
<td>l(letter ell)</td>
<td>To have an integer displayed as a long.</td>
</tr>
</tbody>
</table>
Formatting Characters Supported by the `printf()` function:

Table 4.4 `printf()` conversion characters

<table>
<thead>
<tr>
<th>character</th>
<th>Argument type; displayed as</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, i</td>
<td>int; signed decimal number</td>
</tr>
<tr>
<td>o</td>
<td>int; unsigned octal number (without a leading zero)</td>
</tr>
<tr>
<td>x</td>
<td>int; unsigned hex number using abcdef for 10...15</td>
</tr>
<tr>
<td>X</td>
<td>int; unsigned hex number using ABCDEF for 10...15</td>
</tr>
<tr>
<td>u</td>
<td>int; unsigned decimal number</td>
</tr>
<tr>
<td>c</td>
<td>int; single character</td>
</tr>
<tr>
<td>s</td>
<td>char *; display from the string until a '\0' (NULL)</td>
</tr>
<tr>
<td>p</td>
<td>void *; pointer (implementation-dependent representation)</td>
</tr>
<tr>
<td>%</td>
<td>no argument is converted; print a %</td>
</tr>
</tbody>
</table>

Example for outputting a message (Flight simulation):

```assembly
CR    equ     $0D
LF    equ     $0A
printf equ     $EE88

... ldd     #prompt
jsr     [printf,PCR]
...
prompt  db     "Flight simulation",CR,LF,0
```
Example for outputting three numbers $m$, $n$, and $gcd$ along with a message:

CR equ $0D
LF equ $0A
printf equ $F686
...
ldd gcd
pshd
ldd n
pshd
ldd m
pshd
ldd #prompt
jsr [printf,PCR]
leas 6,sp
...
prompt db “The greatest common divisor of %d and %d is %d”, CR, LF, 0
int far GetCmdLine(char *CmdLineStr, int CmdLineLen)

Pointer address: $EE8A

Incoming parameters: a pointer to the buffer where the input string is to be stored and the maximum number of characters that will be accepted by this function.

- This function is used to obtain a line of input from the user.
- The reception of an ASCII carriage return ($0D) terminates the reception of characters from the user.
- The caller of this function normally would output a message so that the user knows to enter a message. The example in the next page illustrates this interaction.
printf equ $EE88
GetCmdLine equ $EE8A
cmdlinelen equ 100
CR equ $0D
LF equ $0A

... 

prompt db "Please enter a string: ", CR, LF, 0

... 

inbuf ds b 100

... 

ldd #prompt ; output a prompt to remind the user to 
jsr [printf,PCR] ; enter a string 
ldd #cmdlinelen ; push the CmdLineLen
pshd ; "
ldd #inbuf

call [GetCmdLine,PCR] ; read a string from the keyboard
puld ; clean up the stack
Example 4.15 Write a program that invokes appropriate functions to find the prime number between 1000 and 2000. Output eight prime numbers in one line. To do this, you will need to

1.-- write a subroutine to test if an integer is a prime.
2.-- invoke the `printf()` function to output the prime number.
3.-- write a loop to test all the integers between 1000 and 2000.

Solution: The logic structure of the program is

Step 1
Output the message “The prime numbers between 1000 and 2000 are as follows:”.

Step 2
For every number between 100 and 1000 do

1. - call the `test_prime()` function to see if it is a prime.
2. - output the number (call `printf()`) if it is a prime.
3. - if there are already eight prime numbers in the current line, then also output a carriage return.
The algorithm of the `test_prime()`

**Step 1**
Let `num`, `i`, and `isprime` represent the number to be tested, the loop index, and the flag to indicate if `num` is prime.

**Step 2**
isprime ← 0;

**Step 3**
For `i = 2` to `num/2` do
  if `num % i = 0` then
    return;
  isprime ← 1;
return;

The Stack frame of the function `test_prime()`:

Figure 4.18 Stack frame for the prime test subroutine
org $1000
out_buf ds.b 10
PRIcnt ds.b 1
k ds.b 2
tmp ds.b 2
org $1500
start ldx #upper
stx tmp

ldx #lower
stx k ; initialize k to 1000 for prime testing

ldd #form0
jsr [printf,PCR]
lea 4,sp
clr PRIcnt
again ldd k
cpd #upper
bhi Pstop ; stop when k is greater than upper
pshd
ldd #0
pshd
jsr primetest ; test if k is prime
leas 4,sp ; de-allocate space used by outgoing parameters
tsta
beq next_k ; test next integer if k is not prime
inc PRIcnt ; increment the prime count
ldd k
pshd
ldd #form1
jsr [printf,PCR] ; output k
leas 2,sp
ldaa PRIcnt
cmpa #8 ; are there eight prime numbers in the current line?
blo next_k
; output a CR, LF if there are already eight prime numbers in the current line
ldd #form2
jsr [printf,PCR]
clr PRIcnt
next_k ldx k
inx
stx k
lbra again
; Pstop bra $ ; uncomment this line for CodeWarrior
Pstop swi ; comment this line for CodeWarrior
#include "c:\miniide\primetest.asm"
#include "c:\miniide\div32.asm"
#include "c:\miniide\findsqr.asm"
form0 db CR,LF,"The prime numbers between %d and %d are as follows: ",0
form1 db " %d ",0
form2 db " ",CR,LF,0
; org $FFFE ; uncomment this line for CodeWarrior
; dc.w start ; uncomment this line for CodeWarrior
end
Program Execution Result

>g 1500
The prime numbers between 1000 and 2000 are as follows:


PP  PC  SP  X   Y   D = A:B   CCR = SXHI NZVC
30 1560 1500 07D1 15DE 07:D1 1001 0000
xx:1560 34 PSHX
>
Subroutines for Creating Time Delays

Example 4.11 Write a subroutine that can create a time delay of 100 ms.

Solution:

Delay100ms pshx

ldx #60000 ; 2 E cycles
iloop psha ; 2 E cycles
pula ; 3 E cycles
psha ; 2 E cycles
pula ; 3 E cycles
psha ; 2 E cycles
pula ; 3 E cycles
psha ; 2 E cycles
pula ; 3 E cycles
psha ; 2 E cycles
pula ; 3 E cycles
psha ; 2 E cycles
pula ; 3 E cycles
psha ; 2 E cycles
pula ; 3 E cycles
psha ; 2 E cycles
pula ; 3 E cycles
psha ; 2 E cycles
pula ; 3 E cycles
psha ; 2 E cycles
pula ; 3 E cycles
nop ; 1 E cycle
nop ; 1 E cycle
dbne x, iloop
pulx
rts
The previous subroutine can be modified to create a time delay that is a multiple of 100 ms as follows (the multiple is passed in Y):

delayby100ms   pshx
eloop3   ldx  #60000 ; 2 E cycles
iloop3   psha ; 2 E cycles
       pul a ; 3 E cycles
       psha ; 2 E cycles
       pul a ; 3 E cycles
       psha ; 2 E cycles
       pul a ; 3 E cycles
       psha ; 2 E cycles
       pul a ; 3 E cycles
       psha ; 2 E cycles
       pul a ; 3 E cycles
       psha ; 2 E cycles
       pul a ; 3 E cycles
       psha ; 2 E cycles
       pul a ; 3 E cycles
       psha ; 2 E cycles
       pul a ; 3 E cycles
       nop ; 1 E cycle
       nop ; 1 E cycle
dbne x,i loop3 ; 3 E cycles
dbne y,eloop3 ; 3 E cycles
pulx
rts
Introduction to Parallel I/O Port and Simple I/O Devices

- A HCS12 device may have from 48 to 144 pins arranged in 3 to 12 I/O Ports.
- An I/O port consists of a set of I/O pins and the registers required to control its operation.
- An I/O pin can be configured for input or output.
- An I/O pin usually serves multiple functions. When it is not used as a peripheral function, it can be used as a general-purpose I/O pin.

Addressing the I/O Port Data Register

- When inputting or outputting data from or to an I/O port, the user read from or write to the port data register.
- Each I/O port register is assigned to an address in the HCS12 memory space. For example, Port A data register is assigned to address 0, the next instruction output $35 to Port A:

  ```
  movb #$35,0 ; address 0 is Port A data register
  ```

- A name is also assigned to each register so that we can access a register by referring to its name:

  ```
  PTA equ 0
  movb #$35,PTA ; output $35 to Port A
  ```

The name and address association is established by including the HCS12.inc file to your program using the “#include c:\...\hcs12.inc” statement.
Configuring I/O Pin Direction

- Each I/O port has a data direction register DDRx (x is the port name) that configures the direction of each pin.
- By setting a bit of the DDRx register to 1, the corresponding I/O pin is configured for output.
- By setting a bit of the DDRx register to 0, the corresponding I/O pin is configured for input.

```assembly
movb #$FF,DDRB ; configure Port A for output
movb #0,DDRB ; configure Port B for input
movb #$AA,DDRB ; configured Port A odd pins for output, even pins for input
```

Input and Output Operations

```assembly
movb #$FF,DDRB ; configure Port B for output
staa PTB ; output the contents of A to port B
movb #$67,PTB ; output the value $67 to Port B

movb #0,DDRC ; configure Port C for input
movb PTC,ibuf ; read the contents of Port C and save it at the memory
               ; location represented by ibuf
```
The HCS12 I/O ports and Pin Names

Table 4.5 Number of pins available in each parallel port

<table>
<thead>
<tr>
<th>Port Name</th>
<th>No. of Pins</th>
<th>Pin Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8</td>
<td>PA7~PA0</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td>PB7~PB0</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>PE7~PE0</td>
</tr>
<tr>
<td>H</td>
<td>8</td>
<td>PH7~PH0</td>
</tr>
<tr>
<td>J</td>
<td>4</td>
<td>PJ7~PJ0</td>
</tr>
<tr>
<td>K</td>
<td>7</td>
<td>PK4~PK0</td>
</tr>
<tr>
<td>M</td>
<td>8</td>
<td>PM7~PM0</td>
</tr>
<tr>
<td>P</td>
<td>8</td>
<td>PP7~PP0</td>
</tr>
<tr>
<td>S</td>
<td>8</td>
<td>PS3~PS0</td>
</tr>
<tr>
<td>T</td>
<td>8</td>
<td>PT7~PT0</td>
</tr>
<tr>
<td>PAD1, PAD0</td>
<td>16</td>
<td>PAD15~PAD0</td>
</tr>
<tr>
<td>L</td>
<td>8</td>
<td>PL7~PL0</td>
</tr>
<tr>
<td>U</td>
<td>8</td>
<td>PU7~PU0</td>
</tr>
<tr>
<td>V</td>
<td>8</td>
<td>PV7~PV0</td>
</tr>
<tr>
<td>W</td>
<td>8</td>
<td>PW7~PW0</td>
</tr>
</tbody>
</table>
Interfacing with LEDs

- An LED has an **anode** and **cathode** terminal.
- The anode terminal must be at a voltage at least 1.6 V above that of the cathode terminal (forward biased) in order for the LED to be lighted.
- The forward current required to light an LED is from a few to more than 10 mA.
- The I/O pin can drive an LED using one of the circuits shown in Figure 4.15.
- The resistors R1, R2, are R3 are called **current-limiting resistors**.

![Figure 4.15 An LED connected to a CMOS inverter through a current-limiting resistor.](image-url)
LED Circuit in the Draong12-Plus Demo Board

Example 4.12 Write a program to drive the LEDs shown in Figure 4.16 so that one LED is lighted at a time from top to bottom and then from bottom to top with each LED lighted for about 200 ms.
```assembly
#include "C:\miniide\hcs12.inc"

org $1000
lpcnt ds.b 1
org $1500

start movb #$FF,DDRB ; configure port B for output
    bset DDRJ,$02 ; configure PJ1 pin for output
    bclr PTJ,$02 ; enable LEDs to light
forever movb #16,lpcnt ; initialize LED pattern count
    ldx #led_tab ; Use X as the pointer to LED pattern table
led_lpmovb 1,x+,PORTB ; turn on one LED
    ldy #5 ; wait for half a second
    jsr delayby100ms ; "
    dec lpcnt ; reach the end of the table yet?
    bne led_lpmovb
bra forever ; start from beginning

led_tab dc.b $80,$40,$20,$10,$08,$04,$02,$01
dc.b $01,$02,$04,$08,$10,$20,$40,$80
#include "C:\miniide\delay.asm"

; org $FFFE ; uncomment this line for CodeWarrior
; dc.w start ; uncomment this line for CodeWarrior
end
```
Interfacing with Seven-Segment Displays

- Seven-segment displays are mainly used to display decimal digits and a small set of letters.
- The HCS12 I/O port can drive a seven-segment display directly.
- Buffer chips are used mainly to save excessive current draw from the HCS12.
- An example circuit for interfacing with seven-segment display is shown in Figure 4.17.
- Some people may use PB0 to PB6 to drive segment a to g instead.
- The microcontroller must send an appropriate value to the output in order to display a certain value.

![Interfacing with Seven-Segment Displays](image-url)
### Table 4.7 Decimal to seven-segment decoder

<table>
<thead>
<tr>
<th>Decimal digit</th>
<th>Segments</th>
<th>Corresponding Hex Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a b c d e f g</td>
<td>Figure 4.17 circuit Dragond2 demo board</td>
</tr>
<tr>
<td>0</td>
<td>1 1 1 1 1 1 0</td>
<td>$7E$</td>
</tr>
<tr>
<td>1</td>
<td>0 1 1 0 0 0 0</td>
<td>$30$</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0 1 1 0 1</td>
<td>$6D$</td>
</tr>
<tr>
<td>3</td>
<td>1 1 1 1 0 0 1</td>
<td>$79$</td>
</tr>
<tr>
<td>4</td>
<td>0 1 1 0 0 1 1</td>
<td>$33$</td>
</tr>
<tr>
<td>5</td>
<td>1 0 1 1 0 1 1</td>
<td>$5B$</td>
</tr>
<tr>
<td>6</td>
<td>1 0 1 1 1 1 1</td>
<td>$5F$</td>
</tr>
<tr>
<td>7</td>
<td>1 1 1 0 0 0 0</td>
<td>$70$</td>
</tr>
<tr>
<td>8</td>
<td>1 1 1 1 1 1 1</td>
<td>$7F$</td>
</tr>
<tr>
<td>9</td>
<td>1 1 1 1 0 1 1</td>
<td>$7B$</td>
</tr>
</tbody>
</table>

### Displaying Multiple Seven-Segment Displays

- A time-multiplexing technique is often used to display multiple digits.
- The circuit in Figure 4.18 can display up to six digits simultaneous using the time-multiplexing technique.
- The HCS12 lights one digit for a short period of time and then switches to the next digit.
- Within one second, each digit is lighted in turn many times. Due to the **persistence of vision**, all six digits appear to be lighted at the same time.
To display 7 on the display #5

```assembly
movb  #$FF,DDRB ; configure Port B for output
movb  #$3F,DDRP ; configure Port P for output
movb  #$1F,PTP ; enable display #5 to be lighted
movb  #$07,PTB ; send out the segment pattern of 7
```
Example 4.14 Write a program to display 123456 on the six seven-segment displays shown in Figure 4.18.

Solution: The values 1,2,3,4,5, and 6 are displayed on display #5, #4, …, #0, respectively. A table is set up to control the pattern and digit selection as shown in Table 4.8.

<table>
<thead>
<tr>
<th>Seven-Segment Display</th>
<th>Displayed BCD Digit</th>
<th>Port B</th>
<th>Port P</th>
</tr>
</thead>
<tbody>
<tr>
<td>#5</td>
<td>1</td>
<td>$06</td>
<td>$1F</td>
</tr>
<tr>
<td>#4</td>
<td>2</td>
<td>$5B</td>
<td>$2F</td>
</tr>
<tr>
<td>#3</td>
<td>3</td>
<td>$4F</td>
<td>$37</td>
</tr>
<tr>
<td>#2</td>
<td>4</td>
<td>$66</td>
<td>$3B</td>
</tr>
<tr>
<td>#1</td>
<td>5</td>
<td>$6D</td>
<td>$3D</td>
</tr>
<tr>
<td>#0</td>
<td>6</td>
<td>$7D</td>
<td>$3E</td>
</tr>
</tbody>
</table>

```
#include "c:\miniide\hcs12.inc"
org $1500
start lds #$1500
    movb #$FF,DDR B
    movb #$3F,DDR P
    forever idx #DispTab ; set X to point to the display table
    loopi movb 1,x+,PTB ; output segment pattern
    movb 1,x+,PTP ; output display select
    ldy #1
    jsr delayby1ms ; wait for 1 ms
```
Generating a Digital Waveform Using an I/O Pin

- A square wave can be generated by pulling a pin to high for certain amount of time and then pull it to low for the same amount of time and repeat.
- By connecting the pin to a speaker and making the frequency in the audible range, a sound can be made.
Example 4.15 Write a program to generate a 1-kHz periodic square wave from the PT5 pin.
Solution:

```assembly
#include "c:\miniide\hcs12.inc"
org $1500
start lds #$1500
bset DDRT,BIT5 ; configure PT5 pin for output
forever bset PTT,BIT5 ; pull PT5 pin to high
ldy #10 ; wait for 0.5 ms
jsr delayby50us ;
be lr PTT,BIT5 ; pull PT5 pin to low
ldy #10 ; wait for 0.5 ms
jsr delayby50us ;
bra forever
#include "c:\miniide\delay.asm"
; org $FFFE
; dc.w start
end
```

Example 4.16 Write a program to generate a two-tone siren that alternates between 250 Hz and 500 Hz with each tone lasting for half of a second.
The HCS12/MC9S12 Microcontroller

#include "c:\miniide\hcs12.inc"

org $1500

start lds #$1500
  bset DDRT,BIT5 ; configure PT5 pin for output
forever ldx #250 ; repeat 500 Hz waveform 250 times
tone1 bset PTT,BIT5 ; pull PT5 pin to high
  ldy #1
  jsr delayby1ms
  bclr PTT,BIT5
  ldy #1
  jsr delayby1ms
  dbne x,tone1
  ldx #125 ; repeat 250 Hz waveform for 125 times
tone2 bset PTT,BIT5
  ldy #2
  jsr delayby1ms
  bclr PTT,BIT5
  ldy #2
  jsr delayby1ms
  dbne x,tone2
  bra forever
#include "c:\miniide\delay.asm"
end
Interfacing with DIP Switches

- A dual-in-line package can be connected any port with 8 pins.
- A set of pull-up resistors are needed to pull the voltage to high on one side of the DIP.

![Diagram of DIP Switches Connected to Port A](image)

To read data into accumulator A

```
movb #0, DDRA ; configure port A for input
ldaa PTA ; read into accumulator A
```
**Tips for Program Debugging Involving Subroutine Calls**

What to do when the program gets stuck?

**Step 1**
Find out which subroutine gets stuck by setting a breakpoint immediately after the **jsr** or **bsr** instruction.

**Step 2**
Find out why the subroutine gets stuck.
- Forgetting to restore registers pushed onto the stack before return.
- Forgetting to deallocate local variables before return.
- There are some infinite loops in the subroutine.
- Calling other subroutines that do not return.
General Debugging Strategy

1. Make sure all leaf subroutines work correctly by using the methods described in Section 2.9.
2. Debug intermediate subroutines. Make sure no intermediate subroutines get stuck.
3. Debug the top level program.