Recap

Quiz1
Subroutines, the stack, switches, LEDs

Overview

Pointers
Indexed mode addressing
TCNT (free running 16-bit time)
Introduction to Lab 3
Can we collect data to prove it works?
Input, output, time

Read sections 4.5, 6.1, 6.2, 6.3 and 6.11

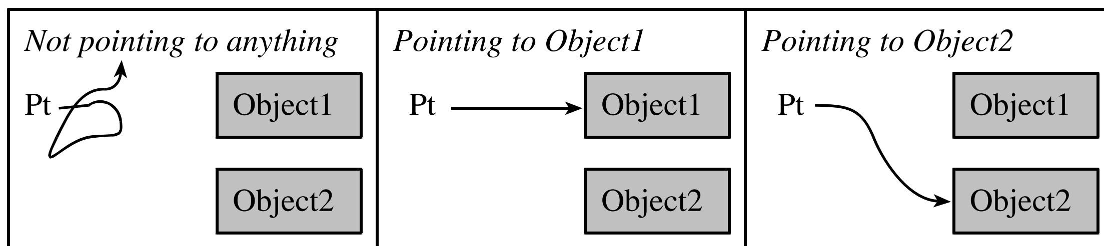


Figure 6.1. Pointers are addresses pointing to objects. The objects may be data, functions, or other pointers.

6.1. Indexed addressing modes used in implement pointers

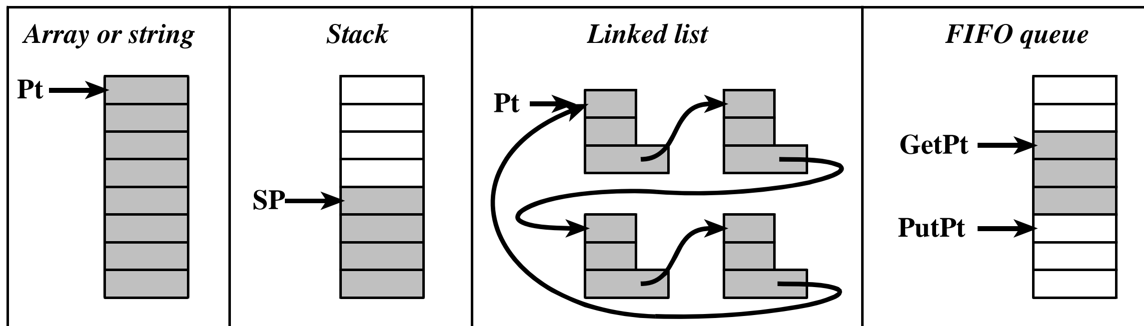
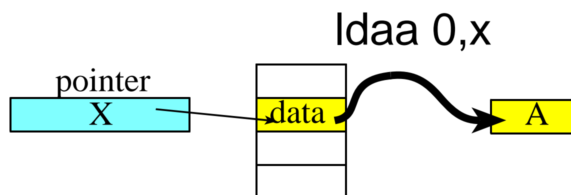


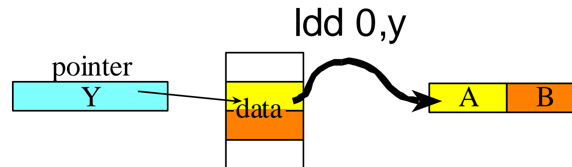
Figure 6.2. Examples of data structures that utilize pointers.

If Register X or Y contains an address, we say it points into memory

```
;read 8-bit contents pointed to by X
```



;read 16-bit contents pointed to by Y



6.1.1. Indexed addressing mode

Indexed addressing uses a fixed offset with the 16-bit registers: X, Y, SP, or PC.

5-bit (-16 to +15),
9-bit (-256 to +127), or
16-bit

machine	opcode	operand	comment
`$6A`**`5C`**	`staa`	`-4,Y`	`[Y-4] = RegA`

Let **`n,R`** be the indexed address

fixed offset **n** and
index register **R** is the index register, then
EAR will be **R**+**n**.

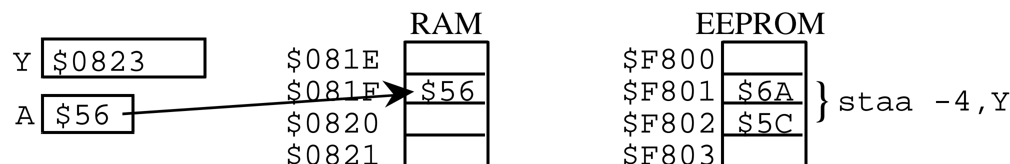


Figure 6.3. Example of the 9S12 indexed addressing mode.

16-bit data structures with indexed addressing is different in assembly versus in C.

```
Prime fdb 1,2,3,5,7,11,13,17,19,23
```

The equivalent ROM-based definition is C would be

```
unsigned short const Prime[10]=
     {1,2,3,5,7,11,13,17,19,23};
```

Want to fetch the 7 from **`Prime[4]`** In assembly,

```
  ldx #Prime    ;pointer to the structure
  ldd 8,x       ;read element number 4
```

or if we could have fetched it directly as

```
  ldd Prime+8   ;read Prime[4]
```

Either way, manipulating addresses in assembly always involves the physical byte-address regardless of the precision of the data.

Want to increment the pointer to the next element.

In C, we define the pointer as

```
unsigned short const *Pt;
```

and initialize it as

```
  Pt =  Prime;
```

To increment the pointer to the next element
in C, use the expression **Pt++**.
In assembly, we can define the pointer in RAM as

```
Pt   rmb 2     ;16-bit pointer to Prime
```

and initialize it as

```
    ldx #Prime
    stx Pt    ;pointer to Prime[0]
```

However, to increment the pointer to the next element we have to add 2 to the pointer. E.g.,

```
    ldx Pt    ;previous pointer
    inx
    inx       ;next element in the 16-bit structure
    stx Pt
```

6.1.2. Auto Pre/Post Decrement/Increment Indexed addressing mode

Post-increment addressing first accesses the data then adds to the index register:

```
    staa 1,Y+ ;Store at 2345, then Reg Y=2346
    staa 4,Y+ ;Store at 2345, then Reg Y=2349
```

Pre-increment addressing first adds to the index register then accesses the data:

```
    staa 1,+Y ;Reg Y=2346, then store at 2346
    staa 4,+Y ;Reg Y=2349, then store at 2349
```

Post-decrement addressing first accesses data then subtracts from index register:

```
    staa 1,Y- ;Store at 2345, then Reg Y=2344
    staa 4,Y- ;Store at 2345, then Reg Y=2341
```

Pre-decrement addressing first subtracts from index register then accesses the data:

```
    staa 1,-Y ;Reg Y=2344, then storeat 2344
    staa 4,-Y ;Reg Y=2341, then store at 2341
```

6.1.3. Accumulator Offset Indexed addressing mode

The offset is located in one of the accumulators A, B or D,
and the base address is in one of the 16-bit registers: X, Y, SP, or PC.

```
    ldab #4
    ldy  #2345
    staa B,Y  ;Store at 2349 (B & Y unchanged)
```

Observation*: Accumulator offset indexed addressing is efficient for accessing arrays. We can place the index in an accumulator and the base address in Reg X or Reg Y.*

6.1.4. Indexed Indirect addressing mode

```
    ldy  #2345
    staa [-4,Y]
;fetch 16-bit address from 2341, store 56 at 1234
```

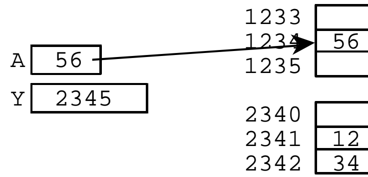


Figure 6.6. Example of the 9S12 indexed-indirect addressing mode.

6.1.5. Accumulator D Offset Indexed Indirect addressing mode

```
    ldd  #4
    ldy  #2341
    stx  [D,Y]
;Store copy of value in Reg X at 1234 (D & Y unchanged)
```

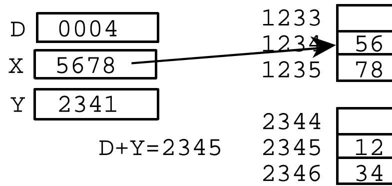


Figure 6.7. Example of the 9S12 accumulator-offset indexed-indirect addressing mode.

6.1.6. Post-byte machine coded for indexed addressing

rr	register
00	X
01	Y
10	SP
11	PC

Table 6.1. The 9S12 indexed-register code.

postbyte(xb)	syntax	mode	explanations
rr000000	,r	IDX	5-bit constant, n=0
rr0nnnnn	n,r	IDX	5-bit constant, n from -16 to +15
rr100nnn	n,+r	IDX	pre-increment, n from 1 to 8
rr101nnn	n,-r	IDX	pre-decrement, n from 1 to 8
rr110nnn	n,r+	IDX	post-increment, n from 1 to 8
rr111nnn	n,r-	IDX	post-decrement, n from 1 to 8
111rr100	A,r	IDX	Reg A accumulator offset
111rr101	B,r	IDX	Reg B accumulator offset
111rr110	D,r	IDX	Reg D accumulator offset
111rr00nnnnnnnnn	n,r	IDX1	9-bit constant, n -256 to 255
111rr010 ffee	n,r	IDX2	16-bit constant, any 16-bit n
111rr111	[D,r]	[D,IDX]	Reg D offset, indirect
111rr011 ffee	[n,r]	[IDX2]	16-bit constant, indirect

Table 6.2. Postbyte values for the 9S12 indexed-addressing modes.

Show xb- table

6.1.6. Load effective address instructions

```
        leax idx ;RegX=EA
        leay idx ;RegY=EA
        leas idx ;RegS=EA
```

In each of the following cases, the effective address, **EA**, is loaded into Register X.

```
leax m,r ;IDX 5-bit index, X=r+m (-16 to 15)
leax v,+r     ;IDX pre-inc, r=r+v, X=r  (1 to 8)
leax v,-r     ;IDX pre-dec, r=r-v, X=r  (1 to 8)
leax v,r+     ;IDX post-inc, X=r, r=r+v  (1 to 8)
leax v,r-     ;IDX post-dec, X=r, r=r-v  (1 to 8)
leax A,r ;IDX Reg A offset, X=r+A, zero padded
leax B,r ;IDX Reg B offset, X=r+B, zero padded
leax D,r ;IDX Reg D offset, X=r+D
leax q,r ;IDX1 9-bit index, X=r+q (-256 to 255)
leax W,r ;IDX2 16-bit index, X=r+W (-32768 to 65535)
```

where **r** is Reg X, Y, SP, or PC, and the fixed constants are

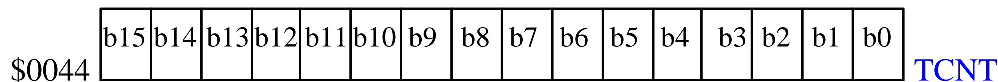
m is any signed 5-bit -16 to +15

q is any signed 9-bit -256 to +255

v is any unsigned 3 bit 1 to 8

W is any signed 16-bit -32768 to +32767 or any unsigned 16-bit 0 to 65535

4.5. 16-bit timer



Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
$0046	**TEN**	TSWAI	TSFRZ	TFFCA	0	0	0	0	TSCR1
$004D	TOI	0	0	0	TCRE	**PR2**	**PR1**	**PR0**	TSCR2
$004F	TOF	0	0	0	0	0	0	0	TFLG2

Table 4.11. 9S12 timer ports.

				E = 8 MHz		E = 24 MHz	
PR2	PR1	PR0	Divide by	TCNT period	TCNT frequency	TCNT period	TCNT frequency
0	0	0	1	125 ns	8 MHz	41.7 ns	24 MHz
0	0	1	2	250 ns	4 MHz	83.3 ns	12 MHz
0	1	0	4	500 ns	2 MHz	167 ns	6 MHz
0	1	1	8	1 µs	1 MHz	333 ns	3 MHz
1	0	0	16	2 µs	500 kHz	667 ns	1.5 MHz
1	0	1	32	4 µs	250 kHz	1.33 µs	667 kHz
1	1	0	64	8 µs	125 kHz	2.67 µs	333 kHz
1	1	1	128	16 µs	62.5 kHz	5.33 µs	167 kHz

Table 4.12. Given an E clock frequency, the PR2 PR1 and PR0 bits define the TCNT rate.

```
; 9S12DP512 at 8 MHz
; Enable TCNT at 1us
Timer_Init
   movb #$80,TSCR1  ;enable
   movb #$03,TSCR2
   rts
```

6.11. Functional Debugging

6.11.1. Instrumentation: dump into array without filtering

Assume **`happy`** is strategic 8-bit variable.

`SIZE equ 20`
`Buf rmb SIZE`
`Pt rmb 2`
	`#define SIZE 20`
	`unsigned char Buf[SIZE];`
	`unsigned char *Pt;`

`Pt` will point into the buffer.
`Pt` must be initialized to point to the beginning, before the debugging begins.

`ldx #Buf` `stx Pt`	`Pt = Buf;`

The debugging instrument saves the strategic variable into the **`Buffer`**.

```
Save
  pshb
  pshx     ;save
  ldx  Pt ;X=>Buf
  cpx  #Buf+SIZE
  bhs  done ;skip if full
  ldab happy
  stab 0,X ;save happy
  inx      ;next address
  stx  Pt
done
  pulx
  pulb
  rts
```

```
void Save(void){
  if(Pt < &Buf[SIZE]){
    (*Pt) = happy;
    Pt++;
  }
}
```

Similar to Program 6.37. Instrumentation dump.

Next, you add **`jsr Save`** statements at strategic places within the system.
Use the debugger to display the results after program is done

6.2. Arrays

Random access
Sequential access.
An **array**
equal precision and
allows random access.
The **precision** is the size of each element.
The **length** is the number of elements (fixed or variable).
The **origin** is the index of the first element.
zero-origin indexing.

Example 6.1. Write a software module to control the read/write (R/W) head of an audio tape recorder. From the perspective shown in Figure 6.8, the stepper motor causes the R/W head to move up and down. This motion affects which audio track on the tape is under the head. The goal is to be able to move the motor one step at a time.

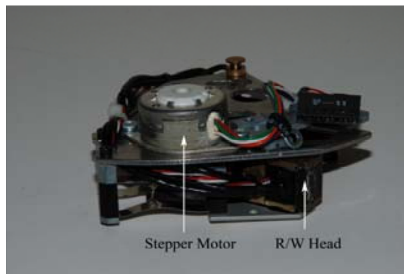


Figure 6.8. A stepper motor is used in a cassette tape recorder to select the track.

Solution

This module requires three public function: one for initialization, one to rotate one step clockwise, and one to rotate one step counter-clockwise. By rotating the motor one step at a time, the software can control which audio track on the tape is under the R/W head. A stepper motor has four digital control lines. To make the stepper motor spin, we output the sequence 5,6,10,9 over and over on these four lines. To make it spin in the other direction, we output the sequence in the other direction. This motor has 24 steps per revolution, therefore one step will change the shaft angle by exactly 15°. To make the motor step once, we output just the next number in the sequence. For example, if the output is currently at 5, and we wish to rotate the shaft by 24°, we simply output a 6. In this solution, we will store the 5,6,10,9 data in an array, as shown in Figure 6.9. For more information on the hardware interfacing of stepper motors see Section 8.7.

Address	Value
$4000	$05
$4001	$06
$4002	$0A
$4003	$09

Figure 6.9. A byte array with 4 elements (addresses are made up to illustrate the array is in ROM).

In C, we can access an element of the array using its name and an index. Assume PTT bits 3-0 are connected to a stepper motor. The initialization function makes those pins an output, and the **Index** is initialized to zero. In assembly, we can perform a similar function using indexed addressing, see Program 6.1. Assume **Index** is an 8-bit private global variable, defined in RAM, and initialized to zero. **Index** takes on the values 0, 1, 2, and 3. The instruction **ldaa B,X** adds the base address in RegX to the index in RegB, fetching the contents of the array at that index. Since the first output generated by **Stepper_CW** will be a 5, we will initialize the motor to 9; this way, the first call to **Stepper_CW** will move the motor. In this example, the subroutine is public but has no input or output parameters. Port T, the array and the index are private to this module. This means if another module wishes to move the motor, it can call the public function **Stepper_CW**, but does not have access to **PTT**, **Data** or **Index**. The third public function, **Stepper_CCW**, is left as Homework problem 6.xx.

```
      org  $0800  ;RAM
Index rmb  1   ;0,1,2,3
      org  $4000  ;ROM
Data  fcb $05,$06,$0A,$09
Stepper_Init
      bset DDRT,#$0F
      movb #$09,PTT
      clr  Index
      rts
Stepper_CW
      ldab Index
      ldx  #Data
      ldaa B,X
      staa PTT  ;24 deg
      incb      ;CW
      andb #$03
```

```
const char static Data[4]=
    {0x05,0x06,0x0A,0x09};
unsigned char static Index;
void Stepper_Init(void){
  DDRT |= 0x0F; // PT3-0 are outputs
  PTT = 0x09;   // first data
  Index = 0;    // first index
}
void Stepper_CW(void){
  PTT = Data[Index];      // rotate 24deg
  Index = 0x03&(Index+1); // next index
}
```

```
       stab Index
       rts
```

Program 6.1. Stepper motor software that uses a byte array.

In general, let **n** be the precision of a zero-origin indexed array in bytes.
If **I** is the index and
Base is the base address of the array,
then the address of the element at **I** is

Base+n*I

In the previous examples, the length of the array was known.
One simple mechanism saves the length of the array as the first element.

```
const char Data[5]={4,0x05,0x06,0x0A,0x09};
const short Powers[6]={5,1,10,100,1000,10000};
```

We could define these variable length arrays in assembly as

```
Data   fcb 4,$05,$06,$0A,$09
Powers fdb 5,1,10,100,1000,10000
```

Another common mechanism to handle variable length is a termination code.

ASCII	code	name
NUL	$00	null
ETX	$03	end of text
EOT	$04	end of transmission
FF	$0C	form feed
CR	$0D	carriage return
ETB	$17	end of transmission block

Table 6.3. Typical termination codes

Checkpoint 6.8: *When accessing sequential data using post-increment indexed mode, how do we select the 1,2,3,4 in* ***1,x+ 2,x+ 3,x+ 4,x+*** *?*

6.3. Strings

A **string** is a data structure with
equal size elements
that only allows sequential access.

Example 6.3. Write software to output a sequence of values to a digital to analog converter.
Solution
In this system, the length of the string is stored in the first byte. This approach is appropriate when the data elements can take on all possible numeric values. Assume a DAC converter is connected to Port T. The function **DAC**, shown in Program 6.5, will output the string data to the DAC. The function uses call by reference, meaning a pointer to the data is passed. The main program calls this function twice, with different data strings.

```
Data1 fcb  4             ;length
      fcb  0,50,100,50   ;data
Data2 fcb  8
      fcb  0,25,50,75,100,75,50,25
*Reg X points to the string data
DAC   ldab 0,x    ;length
loop  inx         ;next element
      ldaa 0,x    ;data
      staa PTT    ;out to DAC
      decb
      bne  loop
```

```
unsigned const char Data1[5]=
  {4,0,50,100,50};
unsigned const char Data2[9]=
  {8,0,25,50,75,100,75,50,25};
void DAC(unsigned char *pt){
unsigned int length;
  length = (*pt++);  // size
  do{
    PTT = (*pt++);
  }
  while(--length);
```

``
       rts
main   lds #$4000
       movb #$FF,DDRT
mloop ldx #Data1 ;first string
       bsr DAC
       ldx #Data2 ;second string
       bsr DAC
       bra mloop
``
	``
	}
	void main(void){
	  DDRT = 0xFF; // outputs to DAC
	  while(1){
	    DAC(Data1); // first string
	    DAC(Data2); // second string
	  }
	}
	``

```
       rts
main   lds  #$4000
       movb #$FF,DDRT
mloop  ldx  #Data1  ;first string
       bsr  DAC
       ldx  #Data2  ;second string
       bsr  DAC
       bra  mloop
```

```c
}
void main(void){
  DDRT = 0xFF;  // outputs to DAC
  while(1){
    DAC(Data1); // first string
    DAC(Data2); // second string
  }
}
```

Program 6.5. A variable length string contains DAC data.

In C, ASCII strings are stored with null-termination. In C, the compiler automatically adds the zero at the end, but in assembly, the zero must be explicitly defined.

Example 6.4. Write software to output an ASCII string to the serial port.
Solution
Because the length of the string may be too long to place all the ASCII character into the registers at the same time, call by reference parameter passing will be used. With call by reference, a pointer to the string will be passed. The function **OutString**, shown in Program 6.6, will output the string data to the serial port. The function **SCI_OutChar** will be developed later in Chapter 8 and shown as Program 8.2. For now all we need to know is that it outputs a single ASCII character to the serial port. The main program calls this function twice, with different ASCII strings.

```
Hello fcc  "Hello World"
      fcb  0
CRLF  fcb  13,10,0
;Reg X points to the string data
OutString
      ldaa 1,x+    ;next data
      beq  done    ;0 means end
      jsr  SCI_OutChar
      bra  OutString
done  rts
main  lds  #$4000
      bsr  SCI_Init
mloop ldx  #Hello  ;first string
      bsr  OutString
      ldx  #CRLF   ;second string
      bsr  OutString
      bra  mloop
```

```c
unsigned const char CRLF[3]=
  {13,10,0};

void OutString(unsigned char *pt){
unsigned char letter;
  while(letter = (*pt++)){
    SCI_OutChar(letter);
  }
}
void main(void){
  SCI_Init();
  while(1){
    OutString("Hello World");
    OutString(CRLF);
  }
}
```

Program 6.6. A variable length string contains ASCII data.

The bottom line

Pointers are addresses
Indexed addressing mode used for pointers
Precision: 8 bit or 16-bit
Arrays and strings have equal precision elements
TCNT is a 16-bit free-running clock